

Program Reduction: Versatility, Insights, and Efficacy

by

Mengxiao Zhang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2025

© Mengxiao Zhang 2025

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner: Tse-Hsun Chen
Associate Professor, Computer Science and Software Engineering
Concordia University

Supervisor(s): Chengnian Sun
Associate Professor, David R. Cheriton School of Computer Science
University of Waterloo

Internal Member: Shane McIntosh
Associate Professor, David R. Cheriton School of Computer Science
University of Waterloo

Internal Member: Michael W. Godfrey
Professor, David R. Cheriton School of Computer Science
University of Waterloo

Internal-External Member: Weiyi Shang
Associate Professor, Electrical and Computer Engineering
University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

I was the primary author for all the research presented in this thesis. My responsibilities included: (1) developing the research ideas, (2) collecting and analyzing data, and (3) drafting the manuscript. My co-authors assisted by refining the research ideas, offering constructive feedback throughout the research process, and helping to improve the manuscript.

In the research presented in § 3, Chengnian Sun mentored me on how to perform the study. Zhenyang Xu helped with the data collection of the project. Yongqiang Tian and Yu Jiang provided feedback in manuscript drafting.

In the research presented in § 4, Chengnian Sun provided supervision in all parts of the research. Xinru Cheng helped with the theoretical analysis, and Zhenyang Xu and Yongqiang Tian provided writing feedback.

In the research presented in § 5, Chengnian Sun supervised the entire study and provided guidance on each part. Yongqiang Tian co-supervised the entire study. Zhenyang Xu, Yiwen Dong and Shin Hwei Tan helped with improving the experiment design and coherence of the manuscript.

Abstract

Given a program P and a property ψ it preserves, the goal of program reduction is to minimize this program while ensuring that the minimized version P_{min} still preserves ψ . Program reduction is a widely used technique to facilitate compiler debugging, as most compiler bugs are triggered by program inputs, which often contain a significant amount of code unrelated to the bug. Although program reduction automates eliminating such bug-irrelevant components from a program, it can take hours to complete due to its trial-and-error strategy. Furthermore, even if the reduction process produces a smaller program after a long wait, the resulting program is not guaranteed to be optimal, as program reduction is an NP-complete problem. As a result, compiler developers consistently seek program reduction approaches that offer faster reduction speeds or produce smaller outputs. It is critical to design superior program reduction approaches to further explore the potential of this field.

This thesis aims to help enhance program reduction approaches in the following three ways.

The first study aims to enhance the versatility of program reduction approaches. While existing techniques, such as C-Reduce and Perses, can effectively reduce a bug-triggering program as a whole, they fail to consider the varied degrees of relevance each remaining element has to the bug. To address this limitation, this study proposes PPR, a new program reduction technique designed to minimize a pair of programs *w.r.t.* certain properties. Given a seed program P_s and its variant P_v with differing properties (e.g., P_v crashes a compiler while P_s does not), PPR produces a pair of minimized programs that preserve these properties separately, with reduced differences highlighting bug-related elements. Evaluation results demonstrate that PPR effectively reduces pairs of programs, producing outputs of comparably small size to those generated by Perses and C-Reduce. In addition, PPR significantly outperforms the Delta Debugging algorithm in isolating bug-related differences.

The second study concentrates on the understanding of a state-of-the-art program reduction approach ProbDD. ProbDD, a variant of ddmin, employs a probabilistic model to estimate the likelihood of each element being relevant to ψ , achieving superior performance compared to the Delta Debugging algorithm. However, the theoretical probabilistic model underlying ProbDD is intricate and not fully understood. To address this, the study conducts the first in-depth theoretical analysis of ProbDD, comprehending and demystifying this state-of-the-art approach from multiple perspectives. Building upon these insights, the study further proposes CDD, a simplified counter-based model that retains the core strengths

of ProbDD. The evaluation demonstrates that CDD can achieve the same performance as ProbDD, despite being much simplified.

The third study integrates large language models (LLMs) to improve the efficacy of program reduction. Existing program reduction techniques typically fall into two categories: generic approaches applicable to a wide range of languages but lacking domain-specific knowledge, and language-specific approaches that exploit detailed language knowledge but fail to generalize across multiple languages. However, effectively combining both language generality and language-specific expertise in program reduction is yet to be explored. To this end, this chapter proposes LPR, the first LLM-aided technique leveraging LLMs to perform language-specific program reduction for multiple languages. The key insight is to synergize the language generality of existing program reducers, such as Perses, with the language-specific semantics learned by LLMs. The evaluation shows that LPR surpasses Vulcan by producing 24.93%, 4.47%, and 11.71% smaller programs, while reducing the reduction time by 10.77%, 34.88%, 36.96% on benchmarks in C, Rust and JavaScript, separately.

Collectively, these studies advance the field of program reduction by enhancing its versatility, insights, and efficacy. By making reduction more effective, faster, and easier to understand, this thesis facilitates efficient debugging and robust compiler development.

Acknowledgements

I would like to thank all those who supported me during my Ph.D. journey. Their support and encouragement made my four years as a researcher in software engineering a more fulfilling experience.

First and foremost, I would like to express my deepest gratitude to my supervisor, Prof. Chengnian Sun. He taught me the essential skills for Ph.D. studies, including academic perspectives, rigor, critical thinking, writing, and more. Chengnian has been supportive throughout my journey, he gave me the freedom to explore my interests, provided invaluable feedback on my work, and offered valuable suggestions for my growth.

I sincerely thank my thesis committee members, Prof. Shane McIntosh, Prof. Michael W. Godfrey, Prof. Weiye Shang and Prof. Tse-Hsun Chen, for their time and effort in providing feedback on the thesis.

I would also thank all co-authors in my publications. The collaboration with them broadened my research view and enhanced my ability. I would like to thank all reviewers for their constructive criticism and suggestions, which further improved my work.

I am deeply grateful for the support from my labmates at the University of Waterloo. Special thanks go to Yongqiang Tian, Yiwen Dong, Zhenyang Xu, Xueyan Zhang, Gaosen Zhao, Boren Zang, Raymond Chang, and Yiran Wang. They provided invaluable help in my studies and research. I sincerely wish them great success in their careers.

I would like to express my heartfelt gratitude to my landlords, Xiaojuan Wang and Jie Cao, for providing me with a comfortable and secure place to live throughout my four years of Ph.D. studies. Their guidance and advice in daily life were immensely helpful, and I am truly grateful for their support.

Finally, I would like to express my deepest gratitude to my parents, Lifeng Zhang and Hongbin Yang, for their unwavering support and encouragement throughout my Ph.D. journey. Over these four years, I shared with them the joy of success and the pain of setbacks. During the challenging times of the COVID-19 pandemic, their constant encouragement gave me the strength and motivation to keep moving forward.

Dedication

This thesis is dedicated to all who share a passion for this field.

Table of Contents

Examining Committee	ii
Author's Declaration	iii
Statement of Contributions	iv
Abstract	v
Acknowledgements	vii
Dedication	viii
List of Figures	xiii
List of Tables	xv
1 Introduction	1
1.1 Motivation	1
1.2 Overview	2
1.3 Contribution	5
1.4 Organization	5

2	Background	6
2.1	Application Scenarios	6
2.2	Three Levels of Program Reduction	7
2.2.1	List-Based Reduction: The Basic Unit of Reduction	7
2.2.2	Tree-Based Reduction: Utilizing Structural Information	8
2.2.3	Semantics-Aware Reduction: Going Beyond Syntax	9
3	PPR: Pairwise Program Reduction	10
3.1	Introduction	10
3.2	Motivation	14
3.2.1	Practicality	15
3.3	Preliminaries and Formalization	16
3.3.1	Classical Program Reduction	16
3.3.2	Differences Between a Pair of Programs	17
3.3.3	Pairwise Program Reduction	18
3.4	Approach	18
3.4.1	Minimizing Tree-Based Differences	21
3.4.2	Minimizing List-Based Differences	22
3.4.3	Minimizing Commonality of P_s and P_v	23
3.5	Evaluation	23
3.5.1	Experimental Setup	25
3.5.2	RQ1. Effectiveness of PPR	26
3.5.3	RQ2. Efficiency of PPR	27
3.5.4	RQ3. Effectiveness of Each Reducer in PPR	29
3.5.5	RQ4. The Impact of the Order of Reducers	30
3.6	Discussion	31
3.6.1	Usefulness of PPR in Fuzzer Design	31
3.6.2	Generality on Other Languages	31

3.6.3	Threats to Validity	33
3.7	Related Work	33
3.8	Conclusion	34
4	Toward a Better Understanding of Probabilistic Delta Debugging	36
4.1	Introduction	36
4.2	Preliminaries	40
4.2.1	The ddmin Algorithm	40
4.2.2	Probabilistic Delta Debugging (ProbDD)	42
4.3	Delving Deeper into Probability and Size	46
4.3.1	On the Probability in ProbDD	46
4.3.2	On the Size of Subsets in ProbDD	49
4.4	Empirical Experiments	49
4.4.1	Benchmarks	50
4.4.2	Evaluation Metrics	50
4.4.3	The Wrapping Frameworks	51
4.4.4	Reproduction Study of ProbDD	51
4.4.5	Impact of Randomness in ProbDD	54
4.4.6	Bottleneck Overcome by ProbDD	54
4.4.7	1-Minimality of ProbDD?	56
4.5	Implications: A Counter-Based Model	57
4.6	Limitations and Threats to Validity	59
4.6.1	Limitations	59
4.6.2	Threats to Validity	60
4.7	Related Work	60
4.8	Conclusion	61

5	LPR: Large Language Models-Aided Program Reduction	62
5.1	Introduction	62
5.2	Background	65
5.2.1	Program Reduction	65
5.2.2	Large Language Models	66
5.3	Approach	66
5.3.1	Motivation	67
5.3.2	Workflow	68
5.3.3	Transformations	70
5.3.4	Multi-Level Prompts	72
5.4	Evaluation	72
5.4.1	Experimental Setup	73
5.4.2	RQ1: Reduction Effectiveness	74
5.4.3	RQ2: Reduction Efficiency	76
5.4.4	RQ3: Effectiveness of Each Transformation	79
5.5	Discussion	81
5.5.1	The Effectiveness of Multi-level Prompt	83
5.5.2	The Impact of Temperature	83
5.5.3	Failures in LPR	84
5.5.4	Threats to Validity	84
5.6	Related Work	86
5.6.1	Program Reduction	86
5.6.2	LLMs for Software Engineering	87
5.7	Conclusion	87
6	Conclusion and Future Work	89
6.1	Summary	89
6.2	Future Work	90
	References	92

List of Figures

3.1	Reduction results by C-Reduce (Fig. 3.1(a)), PPR (Fig. 3.1(b) and Fig. 3.1(c)) and DD (Fig. 3.1(d) and Fig. 3.1(e)) for LLVM-21467. The differences between the seed and variant are highlighted in blue . All programs are slightly adjusted and aligned for clarity. We simplify results from DD for illustration due to the large size of the original version.	11
3.2	Two programs and their tree-based difference. insert , delete and replace from seed and variant are highlighted in blue , red to yellow separately, while the common part is black.	16
3.3	The process of PPR reduces a code snippet of GCC-66691 step by step. Deletion and insertion of tokens are highlighted in red and blue separately. Tokens overlaid by gray are those deleted by PPR.	20
3.4	Final result of GCC-66186 (code snippet). Code highlighted in blue is the final differences from seed to variant.	31
4.1	A running example in Python. Fig. 4.1(a) shows the original program, represented as a list of 8 elements ($l_1, l_2, \dots, l_8$), in which l_8 (<i>i.e.</i> , crash(c)) triggers the crash. Fig. 4.1(b) and Fig. 4.1(c) show the minimized results by ddmin and ProbDD, with removed elements masked in gray . Both minimized programs still trigger the crash. Note that ProbDD cannot consistently guarantee the result in Fig. 4.1(c) and might produce larger results, due to its inherent randomness.	40
4.2	Visualization of queries within ddmin, ProbDD and CDD. In ddmin, three types of queries are displayed via stacked bars, the height of which denotes the query number. Within each bar, the number of successful queries, total queries and the corresponding success rate are annotated.	55

5.1	Code snippet from LLVM-31259, showcasing the original code, the effectiveness of <i>Loop Unrolling</i> , and the final results by LPR, Vulcan and C-Reduce.	67
5.2	The workflow of LPR.	68
5.3	An example of prompt design. ⚙️ and 👤 denote system prompt and user prompt provided by the users. 🗣️ denotes the responses from the LLM. . .	71
5.4	Program size changes induced by each transformation on benchmarks of Benchmark-C. In each subplot, the left box-plot and red dots represent how the size of each program changes before and after executing the transformation. The right box-plot and blue dots represent the size change of each benchmark after executing the transformation and the follow-up Perses reduction. There are a total of 20 benchmarks in Benchmark-C, and each experiment is repeated 5 times. Therefore, we draw 100 data points on each boxplot.	80
5.5	Average size decrease and its percentage induced by each transformation within Benchmark-C.	82
5.6	The number of benchmarks impacted by each transformation within Benchmark-C.	82

List of Tables

3.1	The effectiveness of PPR, PPR_s , DD, Perses and C-Reduce on Benchmark-PPR, including final size and reduction rate. The cells with “0” under DD of Seed Size indicate that the reductions did not finish in 24h, and thus failed to find out a seed. In the last row, reduction rate R does not apply to Seed and Difference under DD, as they are all derived from Variant, and thus there is no original version to compare to.	26
3.2	The execution time of PPR, DD, Perses and C-Reduce on Benchmark-PPR.	28
3.3	Results when disabling a reducer or changing the order of reducers. For each scenario, we list the average final size and reduction rate. The cells with “-” indicates the reduction did not finish in 24h.	30
3.4	Results of PPR and DD on Rust and JavaScript.	32
4.1	The symbols used in this chapter.	40
4.2	Step-by-step outcomes from ddmin on the running example. In each column, a variant is generated and tested against the property ψ . These variants are sequentially generated from left to right. The first row displays the variant identifier, and the second row displays round number r and subset size s . In the following rows, the symbol “✓” denotes an element is included by a certain variant, while gray cells signify that the element have been removed. For the last row, T indicates that the variant still preserves the property ψ , whereas F indicates not.	41
4.3	Step-by-step outcomes from ProbDD on the running example. Similar to Table 4.2, round number, subset size and the details of each variants are presented. For each variant, the probability of each element is noted alongside.	45

4.4	The final size, execution time and query number of ddmin, ProbDD and CDD on BM_C and BM_{DBT} . To address significant variations across benchmarks, the geometric mean rather than the arithmetic mean is employed, providing a smoother measure of the average.	52
4.5	The final size, execution time and query number of ddmin, ProbDD and CDD on BM_{XML} . The last row shows the overall average across all three benchmark suites.	53
5.1	The reduction sizes of Perses, Vulcan, C-Reduce, LPR and LPR+Vulcan. Best results among all algorithms are highlighted in bold font.	75
5.2	The reduction time (in the format of hh:mm:ss) of Perses, Vulcan, C-Reduce, LPR and LPR+Vulcan.	78
5.3	The impact of temperature on reduction sizes.	83
5.4	The reduction sizes of LPR using various LLMs.	86

Chapter 1

Introduction

This chapter introduces the motivation of this thesis, the thesis statement, and the three studies that contribute to this thesis statement. Besides, this chapter also illustrates the contribution and the organization of this thesis.

1.1 Motivation

Compilers play a critical role in software development, as all other software is built upon their use. As one of the most complex software infrastructures in the world, compilers are defect-prone and any bug in compilers may incur significant time and financial losses. Therefore, detecting, localizing and fixing compiler bugs effectively and efficiently is crucial. In recent years, researchers have made significant contributions on program generation and mutation, in hopes of provoking compiler bugs [65, 64, 107, 144, 79, 126].

Generating bug-triggering programs is not the only step in compiler debugging and maintenance. Most programs produced via compiler testing approaches contain a substantial amount of code unrelated to the bug, which may significantly distract compiler developers from identifying the root cause of a bug. For example, studies on GCC and LLVM bugs [108, 159] have shown that the bug-related components can be distilled into around only 30 lines of code on average, in contrast to thousands of lines present in the original programs. As a result, upon finding a bug-triggering program, automatically reducing it to retain only the bug-related components is equally crucial and highly desirable. Given this, compiler developers constantly advocate reducing the size of bug-triggering programs before reporting any bug [37, 82] to simplify debugging efforts and improve efficiency.

However, the general problem of program reduction is NP-complete [86], meaning it relies on a trial-and-error process that is both time-consuming and incapable of guaranteeing optimal results. Even if multiple program reduction approaches [109, 145, 86, 99] have been proposed to address this challenge, they have the significant limitations.

1. Merely reducing the given program may not provide sufficient information about the bug for debugging. Even after reduction, there still might be dozens of lines in the minimized program [18], leaving developers to manually identify which parts are most relevant to the root cause of the bug. However, in the program produced by existing program reduction approaches, not all components are equally relevant to the bug. Therefore, it is highly desirable to propose automated strategies that not only reduce the program but also simultaneously isolate the components most closely related to the bug.
2. The current state-of-the-art variant of the ddmin algorithm, namely ProbDD, demonstrates superior speed compared to other algorithms. However, its underlying probabilistic model has not been fully explored and comprehended by researchers and users, making it difficult to explain its efficiency. Therefore, to enable program reduction researchers and users to better understand its remarkable performance, it is essential to conduct an in-depth analysis on ProbDD, delving into the algorithm’s details and performing comprehensive ablation studies to better understand its mechanisms and performance.
3. Some approaches [142, 109, 86, 48] are designed to be generic, making them applicable to multiple languages but lacking domain-specific knowledge, which limits their ability to achieve smaller results. In contrast, other approaches [99, 92, 55] perform better for certain languages by leveraging semantics of the target languages but struggle to generalize across different languages. Therefore, combining the strengths of both language-generic and language-specific approaches would significantly enhance the efficacy of program reduction.

1.2 Overview

This thesis includes three studies to help enhance program reduction.

Thesis Statement. This thesis improves program reduction by proposing novel and effective techniques, conducting theoretical analysis and providing insightful suggestions. It

enhances the versatility, insights, and efficacy of program reduction through comprehensive exploration of these three aspects.

These three studies support this statement from distinct perspectives. The versatility of program reduction is enhanced by reducing a pair of programs, which provides critical differences as additional information. The understanding of program reduction is deepened through the analysis and simplification of the underlying principles of ProbDD. Meanwhile, the efficacy is improved by integrating LLMs into the program reduction process.

Toward the Versatility of Program Reduction. The first study proposes PPR, a novel program reduction approach designed to effectively minimize pairs of programs while preserving their respective properties. Unlike traditional program reduction methods, which focus solely on reducing a single program, PPR broadens the applicability of program reduction to scenarios involving program pairs. Specifically, given a program pair, P_s and its variant P_v , where P_v triggers a compiler bug while P_s does not, PPR identifies and minimizes the differences between the two programs using list-based and tree-based approaches. After reducing these differences, PPR further shrinks the shared parts of the programs, ultimately producing a minimized pair that retains their distinct properties. Evaluation results demonstrate that, despite targeting program pairs, PPR achieves comparable performance to single-program reduction approaches [109, 99]. Moreover, it significantly outperforms the ddmin algorithm in terms of the reduced program size and the clarity of the insights provided by the differences. This study [151] has been published at the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE’23).

- **PPR: Pairwise Program Reduction**

Mengxiao Zhang, Zhenyang Xu, Yongqiang Tian, Yu Jiang, Chengnian Sun.

The 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE’23), published in 2023.

Toward a Deeper Understanding of ProbDD. The second study provides deeper insights into the state-of-the-art algorithm, ProbDD. This algorithm uses a probabilistic model to predict the next subset of elements to delete from a given list, aiming to maximize the expected number of elements removed. While ProbDD significantly outperforms Delta Debugging in terms of efficiency, its underlying probabilistic model remains not fully understood. To bridge this gap, the study conducts the first in-depth analysis of ProbDD, simplifying the algorithm through mathematical analysis and validating these findings with empirical experiments. This analysis negates the role of randomness in ProbDD,

uncovers the true factors driving its superior performance, and highlights its limitations. As an outcome of this analysis, the study introduces CDD, a simplified algorithm that achieves performance on par with ProbDD while being much easier to understand and implement. This study [150] has been accepted by the 47th International Conference on Software Engineering (ICSE’25).

- **Toward a Better Understanding of Probabilistic Delta Debugging**

Mengxiao Zhang, Zhenyang Xu, Yongqiang Tian, Xinru Cheng, and Chengnian Sun. The IEEE/ACM 47th International Conference on Software Engineering(ICSE’25), accepted in 2025.

Toward the Efficacy of Program Reduction. The third study concentrates on the efficacy of program reduction. Some existing program reducers are generic to a wide range of languages, yet they lack the capacity to further transform the programs based on their language features. Other program reducers, on the contrary, are customized to specific languages, but have limited generality. This study proposes LPR, a novel approach leveraging large language models (LLMs) to synergistically combine language generality and language-specific optimization. The core insight of LPR is to exploit the complementary strengths of language-generic reducers and LLMs. Language-generic reducers efficiently minimize programs to a size manageable for LLMs; in turn, LLMs, trained on the semantics of multiple languages, can accurately identify the language of a given program and perform language-specific transformations, creating new reduction opportunities for the reducers. This iterative process achieves effective and efficient reduction while bridging the gap between language generality and language-specific semantics.

Comprehensive evaluation across 50 benchmarks in C, Rust, and JavaScript demonstrates LPR’s superiority over Vulcan, the state-of-the-art language-generic reducer. LPR produces smaller reduced programs, *i.e.*, 24.93%, 4.47%, and 11.71% smaller for C, Rust, and JavaScript benchmarks, respectively. Moreover, LPR excels in efficiency, reducing large and complex programs faster than Vulcan, with time savings of 10.77%, 34.88%, and 36.96% for C, Rust, and JavaScript benchmarks, respectively. Notably, LPR and Vulcan can complement each other. For instance, applying Vulcan to LPR’s output for C programs achieves results on par with C-Reduce, a highly effective reducer for C.

This study [148] has been published in the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA’24).

- **LPR: Large Language Models-Aided Program Reduction**

Mengxiao Zhang, Yongqiang Tian, Zhenyang Xu, Yiwen Dong, Shin Hwei Tan, Chengnian Sun.

The 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'24), published in 2024.

1.3 Contribution

This thesis makes three main contributions aimed at holistically improving the practical application of program reduction.

Enhancing the Versatility. We extend the functionality of program reduction approaches to facilitate the debugging of compilers. The first study introduces PPR, a novel technique that reduces pairs of programs, providing additional insights beyond traditional single-program reducers.

Deepening the Insights. We conduct an in-depth analysis of ProbDD, a state-of-the-art reduction algorithm, to improve the understanding and optimization of the ddmin algorithm. The second study uncovered the key factors behind ProbDD's superior performance, simplified its approach, and offered actionable recommendations to help researchers and users effectively apply the ddmin algorithm and its variant.

Improving the Efficacy. We explore the use of LLMs to improve both the effectiveness and generality of program reducers. The third study leverages the semantic-aware transformation capabilities of LLMs to refine program reduction results, bridging the gap between language-general and language-specific reduction techniques.

1.4 Organization

The remainder of this thesis introduces the details of these studies. Specifically, § 2 introduces the background, including application scenarios and approaches of program reduction. § 3 introduces our proposed pairwise program reduction approach. § 4 details the ProbDD, our in-depth analysis and simplification, together with our proposed CDD and extensive evaluation. The utilization of LLMs is presented in § 5. Finally, § 6 summarizes our contributions and discusses future work.

Chapter 2

Background

This chapter introduces the background related to this thesis, including the application scenarios and approaches of program reduction.

2.1 Application Scenarios

One of the application scenarios of program reduction is to minimize programs that trigger bugs in language processors, such as compilers. Compiler testing aims to identify bugs in language compilers or processors. In recent years, various approaches, such as program generation [106, 144, 112, 73, 80, 102, 11, 122, 67, 79] and program mutation [107, 64, 65, 74, 127, 20, 126, 66, 154, 31] have been developed to achieve this goal. However, these approaches often produce large programs, which are challenging for compiler developers to debug, as most of the code is unrelated to the bug. Therefore, program reduction techniques are essential in compiler testing to minimize these large test inputs while preserving the bug-triggering behavior, making it easier to identify the root cause of the issue.

Another application scenario is software debloating [84, 38, 61, 98, 45, 14, 6, 140, 97, 104, 111]. As software continues to grow in size and complexity, it often includes numerous features and components that are unnecessary for a specific use case. This bloat not only increases storage requirements but also slows down execution. By making the software smaller, debloating can reduce its size and improve performance by eliminating unused or unnecessary code paths. The goal of software debloating is to minimize the size of the software as much as possible while ensuring that the minimized version still preserves the essential functionalities indicated by the user-provided test cases. Program reduction

approaches, such as ddmin [145] and Perses [109], have been applied to achieve software debloating.

2.2 Three Levels of Program Reduction

Recent years have seen the emergence of various program reduction techniques, which can be broadly categorized into three levels. This section provides a brief overview of each. Furthermore, the three studies in this thesis respectively enhance these levels, by deepening the understanding of list-based reduction, improving versatility via tree level differencing, and enabling semantic transformations through large language models.

2.2.1 List-Based Reduction: The Basic Unit of Reduction

Due to its simplicity and ease of implementation, list-based reduction has become a foundational approach in reduction techniques. This technique treats the target to be minimized as a list of elements, such as characters, lines, or tokens, and systematically eliminates elements that are not related to a given property of interest. A prominent family of techniques in this category is Delta Debugging [145], which systematically reduces failure-inducing inputs while preserving their essential properties. Two key algorithms in this family are ddmin and DD, both of which share the same binary search-like approach but address different problems. The most representative algorithm, ddmin, aims to eliminate elements in a given list that are irrelevant to a property. It works by initially dividing the list into halves and progressively subdividing these subsets, attempting to remove or retain each subset individually until no further reduction is possible. On the other hand, DD focuses on minimizing the difference between a list that exhibits the property and a list that does not, ultimately identifying the essential changes responsible for the property.

In recent years, several optimizations have been proposed to enhance the workflow of ddmin [57, 39]. Additionally, researchers [119, 45] have integrated machine learning techniques into ddmin to further improve its performance and efficiency. List-based reduction, particularly ddmin, is not only useful for bug-triggering input minimization but also finds applications in various other scenarios. It has been applied to minimize and isolate bug-related changes [145, 8], perform test suite reduction [42, 43], configuration simplification [157], *et al.* Its versatility and simplicity make it a widely adopted approach for different reduction tasks across various domains.

Despite its wide adoption, list-based reduction still has room for improvement. The probabilistic nature of ProbDD, a state-of-the-art variant of ddmin, had not been well understood, which limited its interpretability and extensibility. To address this, our second study conducts a comprehensive theoretical and empirical analysis of ProbDD, revealing the core principles behind its efficiency. We further propose CDD, a simplified and deterministic variant that retains comparable performance while being easier to understand and implement. A better understanding of ProbDD foundationally strengthens the core unit underlying many reduction workflows, paving the way for further optimizations.

2.2.2 Tree-Based Reduction: Utilizing Structural Information

Although list-based reduction approaches are general and straightforward, they may be ineffective and inefficient in certain scenarios because they treat the input as a single atomic list, ignoring its structural complexity. This limitation becomes apparent when dealing with highly hierarchical inputs, such as programs. To address this, syntax-guided reduction approaches have been proposed.

Guided by the language syntax of the given program, Hierarchical Delta Debugging [86], short for HDD, parses the program input into a tree structure, and performs list-based reduction approaches, *e.g.*, ddmin, on each level of the tree structure. Unlike treating a deeply nested program as a flat list of tokens, HDD leverages the tree structure and regards the program as a list of tree nodes, aiming to remove tree nodes. This strategy reduces the overall search space, resulting in significantly fewer syntactically invalid programs. Building on this, Perses [109] completely avoids generating syntactically invalid programs by formal syntax transformation. Vulcan [142] further advances the capabilities of Perses by heuristically manipulating tree nodes, pushing the boundaries of syntax-guided reduction.

While syntax-guided reduction ensures syntactic validity, its tree-based structure also enables precise matching between two programs’ parse trees. By leveraging this structure, our first study proposes Pairwise Program Reduction (PPR), a novel approach that computes tree differences between a bug-triggering program and its non-bug-triggering variant. Through identifying changes between a pair of programs, PPR not only reduces the program size but also isolates the critical changes responsible for the bug. Compared to traditional single-program reduction, this approach introduces an additional insight, *i.e.*, critical change isolation, thereby enhancing the versatility of program reduction.

2.2.3 Semantics-Aware Reduction: Going Beyond Syntax

In contrast to syntax-guided reduction, which relies solely on the syntax of a language and can therefore be applied to multiple languages, some reduction approaches are tailored to specific languages by leveraging the semantics for enhanced performance.

For example, C-Reduce [99] incorporates transformation passes such as function inlining to further refine reduction results in C/C++. Similarly, J-Reduce [55, 56] analyzes and simplifies class inheritance relationships of languages like Java, and achieves enhanced simplification. ddSMT [92] is specifically designed to reduce programs in the SMT-LIBv2 format. By utilizing language-specific features, these approaches achieve superior reduction performance.

Although semantics-aware reduction can yield high-quality results, existing methods often trade generality for specialization. For example, tools like C-Reduce rely on manually implemented, language-specific transformation passes, which are powerful but difficult to extend beyond a single language. To bridge this gap, our third study explores the use of Large Language Models (LLMs) in program reduction. We propose LPR, a reduction framework that combines language-general reducers with semantically informed transformations powered by LLMs. Unlike traditional reducers, LPR does not require manually crafted transformation rules; instead, it leverages the cross-language knowledge embedded in LLMs to automatically identify and apply meaningful simplifications. This hybrid approach enables effective reductions across diverse programming languages, significantly improving both the generality and efficacy of program reduction.

Chapter 3

PPR: Pairwise Program Reduction

This chapter introduces Pairwise Program Reduction (PPR), a technique that reduces a pair of programs simultaneously. While traditional single-program reduction focuses solely on minimizing program size, PPR not only achieves size reduction but also isolates critical changes that are closely related to the bug. By leveraging the contrast between two program versions, PPR provides richer diagnostic information, thereby enhancing the versatility of program reduction.

3.1 Introduction

Most reduction techniques reduce a program as a whole and assume that every part of the minimized program is equally relevant to the bug, but overlook the fact that the degree of relevance of each program element to the bug varies. Moreover, mutation-based testing [107, 64, 65, 91, 51, 76, 29, 9, 124] is a widely used approach in compiler testing. It generates new test programs, referred to as variants, by applying random mutations to existing programs, referred to as seeds, in hopes of provoking compiler bugs. The availability of these seed-variant pairs may offer additional insights into understanding and isolating failure-inducing program elements.

For instance, though the state-of-the-art technique C-Reduce can minimize the original bug-triggering program of LLVM-21467 [18] to 103 tokens as shown in Fig. 3.1(a), it can still take compiler developers large efforts to debug, as the critical bug-triggering tokens are not highlighted. In this example, certain tokens are indeed more relevant to the bug than the others: the bug is triggered by code highlighted in Fig. 3.1(c), which modifies the

same variable **b** in multiple basic blocks and is more relevant to the bug than statements such as function calls to **div** and **id**.

<pre>1 #include <stdint.h> 2 uint8_t g; 3 int16_t 4 (div)(int16_t a, int16_t b) 5 { 6 return b == 0 ? a / b; 7 } 8 int64_t (id)() { return 0; } 9 int8_t fn(a) { 10 int32_t d = 1; 11 uint8_t b; 12 for (;;) { 13 uint16_t c = id(b = g); 14 if (div(c, 15 div(10, 65535))) 16 b = g; 17 else 18 d &= a; 19 b = g; 20 } 21 } 22 23 int main() { fn(0); }</pre>	<pre>1 #include <stdint.h> 2 uint8_t g; 3 int16_t 4 div(int16_t a, int16_t b) 5 { 6 return b == 0 ? a : a / b; 7 } 8 int64_t id(a) { return a; } 9 int8_t fn(a) { 10 uint8_t b; 11 for (;;) { 12 uint16_t c = 65532UL; 13 id(b = g); 14 if (div(c, 15 div(0x2FAAL, 65535UL))) 16 a &= 1L; 17 } 18 } 19 20 } 21 22 } 23 int main() { fn(0); }</pre>	<pre>1 #include <stdint.h> 2 uint8_t g; 3 int16_t 4 div(int16_t a, int16_t b) 5 { 6 return b == 0 ? a : a / b; 7 } 8 int64_t id(a) { return a; } 9 int8_t fn(a) { 10 uint8_t b; 11 for (;;) { 12 uint16_t c = 65532UL; 13 id(b = g); 14 if (div(c, 15 div(0x2FAAL, 65535UL))) 16 b = g; 17 else { 18 a &= 1L; 19 } 20 b = g; 21 } 22 } 23 int main() { fn(0); }</pre>
(a) By C-Reduce.	(b) <i>Seed</i> by PPR.	(c) <i>Variant</i> by PPR.
<pre>..... int8_t fn() { } int main() { fn; }</pre>	<pre>..... int8_t fn() { } int main() { fn(); }</pre>	
(d) <i>Seed</i> by DD.		(e) <i>Variant</i> by DD.

Figure 3.1: Reduction results by C-Reduce (Fig. 3.1(a)), PPR (Fig. 3.1(b) and Fig. 3.1(c)) and DD (Fig. 3.1(d) and Fig. 3.1(e)) for LLVM-21467. The differences between the seed and variant are highlighted in blue. All programs are slightly adjusted and aligned for clarity. We simplify results from DD for illustration due to the large size of the original version.

Program Isolation. As shown above, isolating failure-inducing program elements can facilitate debugging. To the best of our knowledge, there are two prior program isolation

techniques.

DD DD [145] is a variant in the Delta Debugging algorithm family. It is worth noting that the original Delta Debugging paper [145] proposed two algorithms, *i.e.*, DD and ddmin. ddmin removes from the given list the elements that are irrelevant to the property. By contrast, DD aims to isolate failure-inducing changes by narrowing the difference between a given list that has a property and an empty list. Given P , it does not only reduce P but also highlights the key failure-inducing elements in P . It returns a pair of programs: the minimized bug-triggering program (referred to as *variant*) reduced from P and the maximized non-bug-triggering program (referred to as *seed*) grown from an empty program.

However, DD does not respect the syntactical constraints of programs and thus does not perform well on programs: the reduction time can be excessively long because of generating large numbers of syntactically invalid programs and the results are usually too large for debugging [86, 109]. Moreover, DD does not use the original seed program, from which P is derived, to find the failure-inducing program elements, and thus the difference between the seed and variant is usually irrelevant to the root cause of the bug. Figs. 3.1(d) and 3.1(e) show the reduced result by DD for LLVM-21467: DD highlights that if the function call `fn` is removed from the variant, then the bug disappears. Though the bug is indeed triggered by certain code in `fn`, the highlighted difference by DD is far less informative than the difference in Fig. 3.1(c). Moreover, in the reduced variant by DD, `fn` has 4,332 tokens, which contains excessive irrelevant information and thus complicates debugging (more details are in § 3.2).

Spirv-fuzz Spirv-fuzz [31] is a testing technique for SPIR-V intermediate representation [132]. Given a seed program P_s , it applies a sequence of random mutations on P_s to generate a variant for testing SPIR-V processors. Different from classical mutation-based testing, Spirv-fuzz intentionally makes each mutation as small and independent as possible. Once a variant P_v finds a bug, ddmin is used to reduce the sequence of mutations that is applied on P_s to generate P_v , so that the bug-irrelevant mutations can be removed; in the end, P_v and the sequence of bug-relevant mutations are included in the bug report to facilitate debugging.

Although Spirv-fuzz can highlight the critical changes and has been adopted at Google [30], several weaknesses limit its application to testing other language implementations, and motivate this work. First, Spirv-fuzz relies on a special mutation-base technique: the mutation operations need to be small and independent, and the sequence of mutations applied to a seed needs to be recorded, modifiable, and replayed. Second, it can merely minimize the differences between the seed and variant, but is not capable of reducing the common part: The authors also acknowledged this limitation and that large seed

programs might diminish the comprehensibility of the bug reports, as quoted from [31]: “The comprehensibility of these bug reports and regression tests depends on a reasonably small original program”.

Despite the importance of program reduction and program isolation in debugging, there is no technique yet that synergistically reconciles both. To this end, we propose Pairwise Program Reduction (PPR), a novel and effective program reduction technique considering both program reduction and failure-inducing change isolation at the same time. Specifically, PPR compares a seed P_s and a bug-triggering variant P_v generated from P_s , and then reduces the *commonalities* and the *differences* of P_s and P_v simultaneously. The reduction result includes a minimized seed P_s^* and a minimized bug-triggering variant P_v^* with the differences between P_s^* and P_v^* minimized as well.

We strongly believe that PPR addresses the shortcomings of DD and Spirv-fuzz, and is practical in at least the following two aspects:

For Developers of Compilers. Debugging compiler bugs is arduous, especially debugging miscompilation faults. It should be noted that a miscompilation is a category of compiler bugs, where the buggy compiler silently compiles a well-defined program into a binary that has unexpected semantics. The minimized differences between P_s^* and P_v^* highlight the minimal required changes from a passing program to a bug-triggering one to induce the bug. Such minimal changes, together with the minimized bug-triggering program, offer a good starting point to compiler developers for fault localization and analysis.

For Researchers on Mutation-Based Compiler Testing. Isolating the critical changes provides insights to researchers on compiler testing, who frequently need to analyze the efficacy of mutations (*e.g.*, which mutation operator is more effective in triggering compiler bugs) to understand and tune their fuzzing techniques. Prior mutation-based research work indeed conducted such analysis, though manually. For example, researchers distilled and posted critical code mutations that contributed to a bug [107, 124, 9] in their paper. However, such manual efforts are often labor-intensive and error-prone, and more importantly can be automated.

We have conducted extensive evaluations of PPR, showcasing its practicality and superiority over DD. First, we evaluated PPR on 20 large real-world C compiler bugs. On average, the number of tokens involved in bug-triggering changes is reduced from 27,880 to only 24, and the overall input size is narrowed from 52,712 to 278 tokens. This result is remarkably small compared to 47,850 tokens from DD, and even comparable to results of the state-of-the-art reduction techniques, *i.e.*, Perses [109] and C-Reduce [99]. For efficiency, PPR is faster than DD on average. Besides C/C++ programs, further evaluation shows

PPR’s generality to other languages.

Contributions. We make the following major contributions.

- We propose the novel, general concept of Pairwise Program Reduction for program reduction and failure diagnosis. It complements classical program reduction approaches by identifying critical bug-triggering differences.
- We propose the first algorithm for pairwise program reduction, including three effective and general reducers to reduce both differences and commonalities between a pair of programs.
- To comprehensively evaluate the performance of PPR, we designed benchmarks ranging from small programs via simple mutations to large programs via complex fuzzing techniques.
- To enable future research on this line of research, we released the source code of PPR and the benchmarks publicly [151].

3.2 Motivation

We use LLVM-21467 as a motivating example to illustrate the benefits of using PPR for analyzing compiler bugs, compared to classical program reduction and the program isolation technique DD.

LLVM-21467 is a bug in LLVM-3.4.2. The original bug-triggering variant contains 28,012 tokens in total, and provokes an infinite loop when being compiled by Clang-3.4.2 with the `-O2` flag. As explained in the bug report [83], the optimization pass `FoldOpIntoPhi` folds certain instructions into PHI nodes, for intermediate analysis in SSA form [133]. When processing a variable in a basic block, it tries to invoke itself if the next reachable block has the same variable. When `FoldOpIntoPhi` encounters two mutually reachable basic blocks modifying the same variable, like `b |= g` on in Fig. 3.1(c), an infinite recursion occurs.

Classical Program Reduction (C-Reduce). In this example, the variant is reduced to 103 tokens by C-Reduce shown in Fig. 3.1(a). While this outcome seems favorable at first glance—with over 99% of the code removed—there still remain slightly more than a hundred tokens. Consequently, it becomes challenging for developers to determine which part of the code is more suspicious than the rest.

Pairwise Program Reduction. By contrast, PPR not only minimizes the bug-triggering program in Fig. 3.1(c), but also provides its corresponding, minimized seed for comparison

in Fig. 3.1(b), as well as a minimal set of changes. In this case, reduced seed/variant has 89/100 tokens, comparable to the result by C-Reduce *w.r.t.* size. More importantly, the reduced differences reveal that the change of control flow and multiple assignments to the same variable are relevant to the bug, as highlighted in Fig. 3.1(c). PPR detects all critical changes successfully, offering developers with a better starting point for fault diagnosis.

DD. Results from DD in Fig. 3.1(d) and in Fig. 3.1(e) achieve smaller difference sizes than PPR (The only difference is deleting `()` from a function call `fn()` in the variant). However, such a difference only indicates that the bug is caused by calling `fn`, which provides far less information than PPR. Moreover, reduced seed and variant given by DD (27,830/27,832 tokens) are several orders larger than those from PPR (89/100 tokens)—the final result still has tens of thousands of tokens, and thus are difficult to be used for debugging.

This example demonstrates the benefits of PPR in both bug-triggering program reduction and critical changes isolation: compared to C-Reduce, PPR achieves a similar reduction size but provides extra information to ease debugging; compared to DD, PPR finds out small program pairs with differences closer to reveal the essence of the bug.

3.2.1 Practicality

Besides the motivating example, several facts on compiler testing inspire our work and make the concept of PPR practical and general. First, in mutation-based testing, the seed from which a bug-triggering variant is derived is always available. Regardless of how drastically the seed is mutated, there are still unchanged code snippets, making program-differencing algorithms applicable. Second, many changes by mutation are superfluous and irrelevant to the bug, and removing them has no impact on reproducing the bug. Third, the generated variants are usually syntactically valid, and thus can be reduced efficiently and effectively with tree-based program reduction algorithms [86, 109].

Spirv-fuzz used at Google is another strong demonstration of the practicality and significance of minimizing failure-inducing changes for debugging language implementations. Different from Spirv-fuzz, PPR generalizes to any programming language and is fuzzer-independent: it can be generalized to new languages, and does not require a customized mutation-based testing technique to work. Moreover, Spirv-fuzz cannot reduce the commonality and can only minimize the sequence of mutations applied on the seed program, which hinders its usefulness if the seed is large. In contrast, PPR can minimize the seed, the variant, and the differences between the seed and variant in a single reduction process.

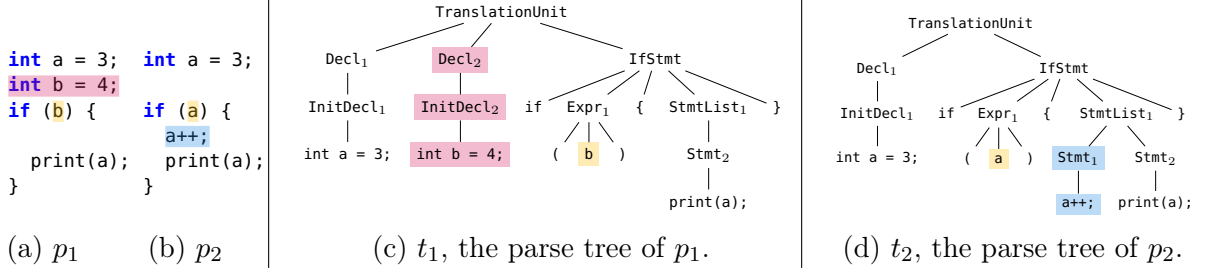


Figure 3.2: Two programs and their tree-based difference. **insert**, **delete** and **replace** from seed and variant are highlighted in **blue**, **red** to **yellow** separately, while the common part is black.

3.3 Preliminaries and Formalization

This section introduces the necessary background and formalizes the problem of pairwise program reduction.

3.3.1 Classical Program Reduction

Given a program P that has a property, let $\mathbb{P}$ denote the universe of all possible programs that are derivable from P by deleting certain tokens ($P \in \mathbb{P}$ as we can delete zero tokens from P). Let $\mathbb{B} = \{\mathbf{True}, \mathbf{False}\}$, then the property of P can be defined as a predicate $\psi : \mathbb{P} \rightarrow \mathbb{B}$; for any program $p \in \mathbb{P}$ (e.g., $p = P$), $\psi(p) = \mathbf{True}$ if p exhibits the property, otherwise $\psi(p) = \mathbf{False}$. As formally defined by Perses [109], the number of tokens in a program p is denoted as $|p|$. The goal of program reduction is to find a program p that satisfies the property and has the fewest tokens in p :

$$\arg \min_{p \in \mathbb{P} \wedge \psi(p)} |p| \equiv \{p | \psi(p) \wedge \forall x \in \mathbb{P}. \psi(x) \wedge |p| \leq |x|\}$$

Program reduction is an NP-complete problem, and all existing algorithms are based on heuristics. Typical such algorithms are:

ddmin. ddmin [145] takes as input a list of elements and a property ψ that the list has, and returns a new list by removing the elements that are irrelevant to ψ from the input list. When applying ddmin to reduce a program with ψ , we can convert the program into a list of lines, tokens, or characters, and let ddmin to remove the ψ -irrelevant elements. However,

ddmin does not consider the syntactical constraints of programs during reduction, making it less effective and less efficient than tree-based program reduction.

Tree-Based Program Reduction. Different from ddmin, syntax-guided program reduction [109, 86, 47, 117] converts the program under reduction into a parse tree [5] *w.r.t.* the formal syntax of the program. Such algorithms traverse the tree in a certain order and repeatedly apply ddmin to reduce a list of tree nodes each time. Due to this tree representation of programs, tree-based program reduction algorithms usually run faster than ddmin and yield smaller results.

C-Reduce. C-Reduce is a widely-used reduction tool customized for C/C++ programs. It translates a program into AST with Clang, and performs a series of source-to-source transformations via LibTooling to simplify the bug-triggering program. C-Reduce can also reduce other programming languages, but many transformations are well-crafted only for C/C++.

3.3.2 Differences Between a Pair of Programs

Program differencing algorithms compute the differences between a pair of programs. Based on how programs are represented for differencing, there are mainly two types of differencing algorithms: list-based [89, 25] and tree-based [33, 94, 36].

List-Based Differencing. A list-based differencing algorithm, denoted as `ListDiff`(l_1, l_2), takes as input two lists, and outputs a list of edit operations (*i.e.*, insert, delete and replace) to edit and transform l_1 to l_2 . For instance, given two programs

$$l_1 = [\text{unsigned}, \text{int}, \text{b}, =, 1, ;]$$

$$l_2 = [\text{int}, \text{b}, , \text{c}, =, 3, ;]$$

`ListDiff`(l_1, l_2) deduces and returns a list of the following four operations: (1) delete `unsigned`, (2) insert `,` after `b`, (3) insert `c` after `b`, and (4) replace `1` with `3`.

Tree-Based Differencing. A tree-based differencing algorithm, denoted as `TreeDiff`(t_1, t_2), infers the differences between two programs by representing programs as trees. It takes as input two trees t_1 and t_2 , and returns a list of operations that converts t_1 to t_2 . For example, given two programs in Fig. 3.2, edit operation list [delete `Decl2`, insert `Stmt1`, replace `b` with `a` on `Expr1`] is returned. More details can be found in [33].

Patching. Patching is an important step to convert a program to another based on differences computed by aforementioned algorithms. Given $d = \text{ListDiff}(l_1, l_2)$, patching is the inverse function, converting l_1 to l_2 with the difference d , *i.e.*, $\text{Patch}(l_1, d) = l_2$.

This algorithm is applicable in two scenarios. First, given d^- , a reduced version of d , $\text{Patch}(l_1, d^-)$ returns l_2^- , closer to l_1 than the original l_2 . Second, if l_1^- is a subset of l_1 , *i.e.*, certain tokens are deleted, $\text{Patch}(l_1^-, d)$ returns l_2^- with those tokens removed and differences applied. The patching process for tree-based differences is similar.

3.3.3 Pairwise Program Reduction

Pairwise program reduction is akin to the classical program reduction but considering a pair of programs. Let P_s and P_v be the original seed and variant programs that exhibit property ψ_s and ψ_v separately, $\mathbb{P}_s$ (and $\mathbb{P}_v$) be the universe of all possible programs derivable from P_s (and P_v) by deleting certain tokens. Let d be the difference between seed and variant programs, $\mathbb{D}$ be the search space over d (*i.e.*, $\mathbb{D}$ includes all possible differences by removing edit operations from d). $|d|$ refers to the number of edit operations in d . The goal of pairwise program reduction is defined as

$$\begin{aligned} PPR(p_s, p_v) \equiv & \\ \{p_s \in \mathbb{P}_s, p_v \in \mathbb{P}_v \mid & \psi_s(p_s) \wedge \psi_v(p_v) \wedge \forall x \in \mathbb{P}_v. \psi(x) \wedge |p_v| \leq |x|\} \\ \text{s.t. } & p_v = \text{Patch}(p_s, d^*) \\ & d^* = \arg \min_{d \in \mathbb{D}} |d| \end{aligned}$$

$p_v = \text{Patch}(p_s, d^*)$ means applying difference d^* to a program p_s to derive a new program p_v . The goal is to minimize both p_v and d . Given that this is a multi-objective optimization problem, for clarity, we prioritize reducing d to d^* , and then minimize p_v with a minimal d^* in the definition and implementation.

3.4 Approach

This section describes PPR in details. Algorithm 1 shows the general workflow of PPR to reduce the seed P_s and the variant P_v *w.r.t.* properties ψ_s and ψ_v . Specifically, P_s exhibits a property ψ_s , *e.g.*, P_s is a well-defined program without undefined behaviors and does not trigger a compiler bug. By contrast, P_v , derived from P_s via mutation, exhibits another property ψ_v . For example, GCC crashes when compiling P_v . The output of PPR is the minimized P_s^* and P_v^* derived from P_s and P_v respectively with $\psi_s(P_s^*) \wedge \psi_v(P_v^*)$.

PPR iteratively and monotonically reduces P_s and P_v until neither of them can be further reduced. In each iteration on line 2 - line 5, PPR reduces the programs in the

Algorithm 1: PPR (P_s, P_v, ψ_s, ψ_v)

Input: P_s : the seed that does not trigger the bug.

Input: P_v : the bug-triggering variant derived from P_s .

Input: $\psi_s : \mathbb{P} \rightarrow \mathbb{B}$: the property to be preserved by P_s .

Input: $\psi_v : \mathbb{P} \rightarrow \mathbb{B}$: the property to be preserved by P_v .

Output: (P_s^*, P_v^*) : the minimized seed and variant with minimal differences.

```
1 repeat
    /* Monotonically minimize the size of  $P_s$ ,  $P_v$  and the differences
    between the two programs. */
2    $P_s^* \leftarrow P_s, P_v^* \leftarrow P_v$ 
    // Minimize tree-based differences.
3    $(P_s, P_v) \leftarrow \text{MinTreeDiff}(P_s, P_v, \psi_s, \psi_v)$ 
    // Minimize list-based differences.
4    $P_v \leftarrow \text{MinListDiff}(P_s, P_v, \psi_v)$ 
    // Minimize the commonality between  $P_s$  and  $P_v$ .
5    $(P_s, P_v) \leftarrow \text{MinCommonality}(P_s, P_v, \psi_s, \psi_v)$ 
6 until  $|P_s| = |P_s^*|$  and  $|P_v| = |P_v^*|$ 
7 return  $(P_s^*, P_v^*)$ 
```

following three steps:

1. PPR calls the reducer **MinTreeDiff** (i.e., Algorithm 2) on line 3 to minimize the tree-based differences between P_s and P_v .
2. PPR calls the reducer **MinListDiff** (i.e., Algorithm 3) on line 4 to minimize the list-based differences between P_s and P_v .
3. PPR calls the reducer **MinCommonality** (i.e., Algorithm 4) on line 5 to reduce the common parts of P_s and P_v .

The first two steps are to minimize the differences between P_s and P_v so that the bug-irrelevant differences can be removed; the third one is to remove the bug-irrelevant program elements on P_s , and remove the corresponding part on P_v indirectly.

A single iteration of calling the three reducers usually cannot reduce the programs to the minimality, since the deletion of some elements may create new reduction opportunities to remove other elements. For example, if all the invocations of a function have been deleted, then it is possible to delete the definition of this function; in other words, the definition of a function can be deleted only if there is no call to the function. Since all the three reducers monotonically decrease the size of programs, the loop is guaranteed to terminate, when neither P_s nor P_v can be further reduced.

<pre> 1 // seed.c 2 for (a = 0; (a != 22); ++a) { 3 int32_t b = 7L; 4 int32_t c = 0x2474AFADL; 5 int32_t d = (-4L); 6 e--; 7 f = g[0][9]; 8 if (b) 9 break; 10 11 }</pre>	<pre> // variant.c 1 for (a = 0; (a != 22); ++a) { 2 int32_t b = 7L; 3 int32_t c = 0x2474AFADL; 4 int32_t d = (-4L); 5 while(b == 0); 6 {{ 7 if (b) 8 ; 9 }} 10 11 }</pre>
---	--

(a) Original programs.

<pre> 1 for (a = 0; (a != 22); ++a) { 2 int32_t b = 7L; 3 int32_t c = 0x2474AFADL; 4 int32_t d = (-4L); 5 e--; 6 f = g[0][9]; 7 if (b) 8 break; 9 10 }</pre>	<pre> 1 for (a = 0; (a != 22); ++a) { 2 int32_t b = 7L; 3 int32_t c = 0x2474AFADL; 4 int32_t d = (-4L); 5 while(b == 0); 6 {{ 7 if (b) 8 ; 9 }} 10 }</pre>
--	--

(b) After calling MinTreeDiff.

<pre> 1 for (a = 0; (a != 22); ++a) { 2 int32_t b = 7L; 3 int32_t c = 0x2474AFADL; 4 int32_t d = (-4L); 5 e--; 6 f = g[0][9]; 7 if (b) 8 break; 9 10 }</pre>	<pre> 1 for (a = 0; (a != 22); ++a) { 2 int32_t b = 7L; 3 int32_t c = 0x2474AFADL; 4 int32_t d = (-4L); 5 while(b == 0); 6 {{ 7 if (b) 8 ; 9 }} 10 }</pre>
--	--

(c) After calling MinListDiff.

<pre> 1 for (a = 0; (a != 22); ++a) { 2 int32_t b = 7L; 3 int32_t c = 0x2474AFADL; 4 int32_t d = (-4L); 5 e--; 6 f = g[0][9]; 7 if (b) 8 break; 9 10 }</pre>	<pre> 1 for (a = 0; (a != 22); ++a) { 2 int32_t b = 7L; 3 int32_t c = 0x2474AFADL; 4 int32_t d = (-4L); 5 while(b == 0); 6 {{ 7 if (b) 8 ; 9 }} 10 }</pre>
--	--

(d) After calling MinCommonality.

Figure 3.3: The process of PPR reduces a code snippet of GCC-66691 step by step. Deletion and insertion of tokens are highlighted in red and blue separately. Tokens overlaid by gray are those deleted by PPR.

To exemplify how the three reducers work, in Fig. 3.3, we use code snippets in GCC-66691 [17] to demonstrate how PPR reduces them step by step. This crash bug in GCC-5.1.0 was discovered with a mutation-based testing technique [107]. As shown in Fig. 3.3(a), the original changes include 25 tokens; but after applying PPR, the deletion of **break** statement is proved to be the only critical change highly relevant to the bug, as highlighted in Fig. 3.3(d).

Algorithm 2: $\text{MinTreeDiff}(P_s, P_v, \psi_s, \psi_v)$

Input: P_s : the seed program.
Input: P_v : the variant program.
Input: $\psi_s : \mathbb{P} \rightarrow \mathbb{B}$: the property to be preserved by P_s .
Input: $\psi_v : \mathbb{P} \rightarrow \mathbb{B}$: the property to be preserved by P_v .
Output: (P_s^*, P_v^*) : the seed and variant program with minimized tree-based differences.

```

1  $d \leftarrow \text{TreeDiff}(P_s, P_v)$ 
2  $P_s^* \leftarrow P_s, P_v^* \leftarrow P_v$ 
3 foreach  $edit \in d$  do
4   switch  $edit$  do
5     case delete a subtree  $t$  from  $P_s$  do
6       // Minimize  $t$  on  $P_s$ , as  $t$  is unique on  $P_s$ .
7        $P_s^* \leftarrow \text{MinSubtree}(P_s^*, \psi_s, t)$ 
8     case insert a subtree  $t$  to  $P_v$  do
9       // Minimize  $t$  on  $P_v$ , as  $t$  is unique on  $P_v$ .
10       $P_v^* \leftarrow \text{MinSubtree}(P_v^*, \psi_v, t)$ 
11     case replace a subtree  $t$  with another subtree  $s$  do
12       // Undo the replacement on  $P_v$ .
13        $P_v' \leftarrow (P_v^* \setminus s) \cup t$ 
14       if  $\psi_v(P_v')$  then  $P_v^* \leftarrow P_v'$ 
15 return  $(P_s^*, P_v^*)$ 

```

3.4.1 Minimizing Tree-Based Differences

Algorithm 2 minimizes the tree-based differences between P_s and P_v . In this process, PPR first deduces the sequence of tree-based edit operations d from P_s to P_v on line 1, and then reduces the subtree involved in each edit operation in d on line 3-line 11. PPR processes edit operations differently based on their types.

- (line 5–line 6) **delete** a subtree t from P_s : PPR attempts to remove nodes of t from P_s by calling `MinSubtree( $P_s^*$ ,  $\psi_s$ ,  $t$ )`, so that the unique subtrees from P_s can be minimized. In Fig. 3.3(b), `e--;` and `f = g[0][9];` on P_s are removed by this step.
- (line 7–line 8) **insert** a subtree t to P_v : Similarly, PPR attempts to remove subtrees of t from P_v to minimize the unique subtree of P_v . In Fig. 3.3(b), `while(b == 0);` on P_v is removed by this step.
- (line 9–line 11) **replace** a subtree t with s in P_v : PPR attempts to undo the replacement by replacing the subtree s with the subtree t .

Minimizing a Subtree. Given a program P , the property ψ of P , and a *subtree* of P , the function `MinSubtree` removes the descendant nodes of *subtree* that are irrelevant to ψ . `MinSubtree` is similar to existing tree-based reduction techniques such as HDD [86] and Perses [109]. The major difference is that both HDD and Perses start the reduction process from the root node of P , whereas `MinSubtree` starts the reduction process from the root node of the given *subtree*. `MinSubtree` can be implemented by extending HDD or Perses to start reduction from a specified tree node.

3.4.2 Minimizing List-Based Differences

Algorithm 3 describes how PPR minimizes list difference between P_s and P_v . At first, the list-based difference d is inferred on line 1. Then PPR uses `ddmin` to minimize d . Note that we simplify this step by enumerating all possible deletions by `ddmin` for illustration purpose on line 3. `ddmin` splits the list into partitions, tries deleting each partition or its complement, and splits each partition into halves for finer-grained deletion. The actual procedure is that every time `ddmin` removes some elements, `Patch` is applied to verify whether the remaining differences still generate a bug-triggering variant, and thus dynamically determine the next step.

Although both tree-based and list-based difference reduction aim at minimizing difference between P_s and P_v , they complement each other. The former is more efficient by leveraging the formal syntax, the latter adopts `ddmin`, which is less effective but simpler and more general. In the example, pairs of redundant curly brackets `{{` and `}}` inserted to P_v are removed (Fig. 3.3(c)). Such changes are neglected by tree differencing algorithms due to the limitation of the used tree differencing algorithms.

Algorithm 3: MinListDiff(P_s, P_v, ψ_v)

Input: P_s : the seed program.
Input: P_v : the variant program.
Input: $\psi_v : \mathbb{P} \rightarrow \mathbb{B}$: the property to be preserved by P_v .
Output: P_v^* : $\psi_v(P_v^*)$, a variant with minimized difference from P_s .

```
1  $d \leftarrow \text{ListDiff}(P_s, P_v)$ 
2  $P_v^* \leftarrow P_v$ 
  // Diff candidates sorted by size in ascending order.
3  $\text{candidates} \leftarrow$  all possible deletions by ddmin
4 foreach  $d \in \text{candidates}$  do
5    $P_v^* \leftarrow \text{Patch}(P_s, d)$ 
6   if  $\psi_v(P_v^*)$  then
7      $\text{break}$ 
8 return  $P_v^*$ 
```

3.4.3 Minimizing Commonality of P_s and P_v

Algorithm 4 shows how PPR reduces the common part of P_s and P_v synchronously. The intuition is to preserve list-based difference d , reduce P_s tentatively, and patch d to each reduced seed candidate to derive the corresponding reduced P_v candidate, and check them against property separately. The difference d , *i.e.*, the edit sequence from P_s to P_v , is deduced and kept on line 1. The root node of P_s is added to worklist on line 3, and then is traversed and reduced on line 4–line 12. PPR uses workflow of tree-based program reduction algorithm to search for smaller candidates of P'_s on line 6. Similar to Algorithm 3, we enumerate all possible candidates for simplification. Each time a P'_s is generated, PPR patches the precomputed d onto it to derive a corresponding P'_v on line 8. If both P_s and P_v satisfy the property, PPR updates the best results, obtains the remaining components *rest*, and pushes them into *worklist* for future traversal.

In Fig. 3.3(d), the common part is reduced, including bug-irrelevant statement `if(b)` and unused variables `b`, `c`, and `d`.

3.5 Evaluation

In this section, we evaluate the effectiveness and efficiency of PPR. Specifically, we investigated the following research questions.

Algorithm 4: MinCommonality $(P_s, P_v, \psi_s, \psi_v)$

Input: P_s : the seed program, must be parsable.

Input: P_v : the variant program.

Input: $\psi_s : \mathbb{P} \rightarrow \mathbb{B}$: the property to be preserved by P_s .

Input: $\psi_v : \mathbb{P} \rightarrow \mathbb{B}$: the property to be preserved by P_v .

Output: (P_s^*, P_v^*) : the directly minimized seed and variant.

```
1  $d \leftarrow \text{TreeDiff}(P_s, P_v)$ 
2  $P_s^* \leftarrow P_s, P_v^* \leftarrow P_v$ 
3  $worklist \leftarrow \{ \text{root node of } P_s \}$ 
4 while  $|worklist| > 0$  do
5    $t \leftarrow \text{Pop}(worklist)$ 
6   // Seed candidates sorted by size in ascending order.
7    $candidates \leftarrow$  all possible deletions on  $t$ 
8   foreach  $(P'_s, rest) \in candidates$  do
9      $P'_v \leftarrow \text{Patch}(P'_s, d)$ 
10    if  $\psi_s(P'_s) \wedge \psi_v(P'_v)$  then
11       $P_s^* \leftarrow P'_s, P_v^* \leftarrow P'_v$ 
12       $\text{Push}(worklist, rest)$ 
13      break
13 return  $(P_s^*, P_v^*)$ 
```

- **RQ1:** Effectiveness of PPR on reduction and isolation.
- **RQ2:** Efficiency of PPR on reduction and isolation.
- **RQ3:** Effectiveness of each individual reducer in PPR.
- **RQ4:** The impact of the order of reducers in PPR.

3.5.1 Experimental Setup

We implement PPR on top of Perses [109]. All experiments are conducted on an Ubuntu 20.04 server with Intel Xeon Gold 5217 CPU@3.00GHz and 384GB RAM.

Benchmark-PPR. Since there is no existing benchmark suite for pairwise program reduction, we constructed Benchmark-PPR to answer the above questions. We collected 20 pairs of seed and variant programs from previous mutation-based C compiler testing studies [107, 65]. According to these studies, these seeds are generated by CSmith [144] and the variants were derived from the seed programs using a sequence of mutations. Table 3.1 shows the information of Benchmark-PPR. The columns **Bug ID** shows the bugs that are triggered by the variant programs. The columns **Original** under **Seed Size**, **Variant Size** and **Difference Size** show the original size of seed, variant and their difference.

These variants triggered different types of bugs in GCC or LLVM, including crash, hang and miscompilation. Moreover, cases in this benchmark suite involve diverse mutations. For instance, LLVM-25900 and GCC-66375 were generated by merely inserting code snippets; GCC-58731 was derived via deletion. Cases like LLVM-22382 involve insertion, deletion and modification. Additionally, the similarity between a seed and a program varies. In GCC-66412, only 3% of the program is mutated, while in LLVM-25900, the final variant only shares 22% in common with the seed.

Baseline. We use DD as the evaluation baseline since it is the only existing tool having the same target as PPR. Spirv-fuzz is not suitable for this benchmark suite as it is customized for a low-level language and it only reduces the difference. Although reducing a pair of programs constraint by differences in between is harder than reducing a single one, we still compare PPR with state-of-the-art classical program reduction algorithms, *i.e.*, Perses and C-Reduce, to demonstrate PPR’s performance is still acceptable.

3.5.2 RQ1. Effectiveness of PPR

We measure the effectiveness of the PPR using the **Reduction Rate** R , *i.e.*, the ratio of tokens in seed/variant/difference that are removed in the reduction. A higher R for seed, variant and difference programs is preferred, as it indicates more bug-irrelevant elements are removed to facilitate debugging process.

Table 3.1: The effectiveness of PPR, PPR_s , DD, Perses and C-Reduce on Benchmark-PPR, including final size and reduction rate. The cells with “0” under DD of Seed Size indicate that the reductions did not finish in 24h, and thus failed to find out a seed. In the last row, reduction rate R does not apply to Seed and Difference under DD, as they are all derived from Variant, and thus there is no original version to compare to.

Bug ID	Seed Size (token #)				Variant Size (token #)						Difference Size (token #)			
	Original	PPR	PPR_s	DD	Original	PPR	PPR_s	DD	Perses	C-Reduce	Original	PPR	PPR_s	DD
LLVM-18556	17,845	212	313	24,723	28,250	279	397	24,725	223	277	10,653	67	84	2
LLVM-18596	31,575	180	524	0	43,890	226	616	37,484	237	223	23,152	46	148	37,484
LLVM-19595	34,898	131	871	0	31,023	138	1,169	26,809	137	119	19,312	7	350	26,809
LLVM-21467	22,798	89	318	27,830	28,012	100	337	27,832	138	103	11,540	11	19	2
LLVM-21582	33,128	486	1,096	33,312	38,775	548	1,348	33,316	639	409	14,010	71	325	4
LLVM-22337	64,803	258	523	63,215	70,545	268	542	63,216	259	210	26,236	10	94	1
LLVM-22382	44,838	100	1,337	16,994	21,361	137	1,407	16,998	117	119	29,455	47	496	4
LLVM-23309	51,404	425	645	34,168	38,920	423	640	34,170	436	350	18,484	16	65	2
LLVM-25900	17,504	196	256	79,224	79,229	226	300	79,229	226	115	61,725	30	46	5
LLVM-26350	36,002	198	253	0	124,058	224	296	115,598	165	221	88,056	26	43	115,598
LLVM-26760	44,250	32	172	199,067	209,824	81	257	199,068	81	59	165,574	49	85	1
GCC-58731	51,285	218	218	0	30,313	213	213	24,654	215	231	20,972	5	5	24,654
GCC-60452	24,259	263	278	14,807	16,452	256	269	14,808	221	233	7,807	7	9	1
GCC-61047	23,043	259	366	16,319	17,179	251	348	16,336	270	64	5,864	8	18	17
GCC-61383	24,478	242	603	26,773	27,017	262	635	26,775	202	201	8,221	20	78	2
GCC-65383	70,001	148	370	42,315	44,215	150	382	42,317	138	83	27,057	28	94	2
GCC-66186	44,373	277	277	45,472	47,708	306	306	45,474	304	297	3,335	29	29	2
GCC-66375	62,375	427	427	0	65,715	429	429	59,102	417	417	3,340	2	2	59,102
GCC-66412	69,849	394	461	70,592	72,000	399	466	70,594	326	361	2,151	5	5	2
GCC-66691	30,408	646	678	15,551	19,753	645	676	15,553	721	570	10,663	1	2	2
Mean	39,956	259	499	35,518	52,712	278	552	48,703	274	233	27,880	24	100	13,185
Mean R		99.35%	98.75%			99.47%	98.95%	7.61%	99.48%	99.56%		99.91%	99.64%	

Columns **PPR** of Table 3.1 show the effectiveness of PPR on Benchmark-PPR. On average, the size of seed and variant is reduced from 39,956 tokens to 259 tokens and from 52,712 tokens to 278 tokens, respectively. Meanwhile, the average difference between seed and variant is reduced from 27,880 tokens to only 24 tokens. The corresponding reduction rates R of PPR for seeds, variants and differences are 99.35%, 99.47% and 99.91%, respectively. These results demonstrate that PPR are effective in reducing the seed, variant and their differences. To show the effectiveness of the main loop in Algorithm 1, we also measure PPR with only a single iteration. As shown in Columns PPR_s , reducing only one iteration produces larger programs and differences compared to multiple iterations.

We also compare PPR with classical program reduction tools, *i.e.*, Perses and C-Reduce. The size of the reduced variant by PPR is 278 tokens on average (varying from 81 tokens to 645 tokens), while the average size of variant by Perses and C-Reduce are 274 and 233, respectively. In 7 and 2 out of 20 cases, PPR even generates smaller programs than Perses and C-Reduce, respectively. This shows that the sizes of variant reduced by PPR are comparable with the ones by classical program reduction such as Perses and C-Reduce. In other words, PPR does not sacrifice the reduction effectiveness on the bug-triggering programs (*i.e.*, variant programs) when effectively reducing the difference between seed and variant programs.

Columns **DD** of Table 3.1 show the effectiveness of DD. On average, DD only achieves 7.61% reduction rate on variant programs. For seed and difference, DD cannot produce acceptable results, and the reduction rate is not applicable. Our explanation is that DD performs a binary search to delete certain consecutive tokens from the variant to derive a subset (seed) that does not trigger the bug, *i.e.*, final seed and differences do not stem from the original ones. Besides, programs with highly nested structure in Benchmark-PPR cannot be effectively handled by DD. Specifically, in 15 out of 20 cases, the difference between the seed and variant programs are reduced to a few tokens, but the size of the seed programs are still more than hundreds of thousands of tokens, making it impossible to comprehend and debug. For the other 5 cases, the seed programs are still zero-token. In other words, by removing consecutive tokens from the variant, DD cannot always find a syntactically valid seed that does not trigger the bug.

In summary, the usefulness of DD in pairwise program reduction is limited, since “a small difference provides insights only if the whole program is small and comprehensible” [31]. By contrast, PPR can effectively reduce the seed and variant programs and minimize the difference between them.

RQ1: PPR generates programs several orders of magnitude smaller than DD, and has a comparable performance to the state-of-the-art tools, *i.e.*, Perses and C-Reduce, in terms of the sizes of the reduced variants.

3.5.3 RQ2. Efficiency of PPR

To answer this research question, we measured the time of PPR in reduction and isolation on Benchmark-PPR, as shown in Table 3.2. To provide developers with in-time results, a short reduction time is preferred. In all 20 cases, PPR is able to finish the reduction and isolation in 24 hours, ranging from 0.84 hour to 18.38 hours. On average, it takes PPR 4.55

Table 3.2: The execution time of PPR, DD, Perses and C-Reduce on Benchmark-PPR.

Bug ID	Time (hour)			
	PPR	DD	Perses	C-Reduce
LLVM-18556	3.03	2.28 (0.75x)	1.55 (0.51x)	4.54 (1.50x)
LLVM-18596	5.41	24.00 (4.44x)	1.62 (0.30x)	4.20 (0.78x)
LLVM-19595	9.62	24.00 (2.49x)	1.41 (0.15x)	3.69 (0.38x)
LLVM-21467	3.35	0.05 (0.01x)	2.15 (0.64x)	4.79 (1.43x)
LLVM-21582	5.72	0.60 (0.11x)	2.21 (0.39x)	3.89 (0.68x)
LLVM-22337	6.41	2.88 (0.45x)	1.54 (0.24x)	2.51 (0.39x)
LLVM-22382	7.54	15.10 (2.00x)	0.19 (0.03x)	0.56 (0.07x)
LLVM-23309	3.16	4.66 (1.47x)	1.32 (0.42x)	3.39 (1.07x)
LLVM-25900	4.26	0.25 (0.06x)	0.79 (0.18x)	1.73 (0.41x)
LLVM-26350	17.66	24.00 (1.36x)	2.85 (0.16x)	3.47 (0.20x)
LLVM-26760	9.59	13.57 (1.42x)	1.50 (0.16x)	1.67 (0.17x)
GCC-58731	1.02	24.00 (23.60x)	0.62 (0.61x)	2.17 (2.13x)
GCC-60452	0.93	0.97 (1.04x)	0.41 (0.44x)	1.61 (1.73x)
GCC-61047	0.80	0.56 (0.69x)	0.47 (0.58x)	1.20 (1.49x)
GCC-61383	2.12	0.12 (0.06x)	0.78 (0.37x)	2.24 (1.06x)
GCC-65383	4.04	0.34 (0.08x)	0.95 (0.23x)	2.43 (0.60x)
GCC-66186	0.85	0.27 (0.31x)	1.00 (1.18x)	2.41 (2.84x)
GCC-66375	2.05	24.00 (11.69x)	1.95 (0.95x)	4.27 (2.08x)
GCC-66412	2.24	0.27 (0.12x)	1.50 (0.67x)	1.66 (0.74x)
GCC-66691	1.18	1.62 (1.38x)	1.08 (0.91x)	2.75 (2.33x)
Mean	4.55	8.18 (2.68x)	1.29 (0.46x)	2.76 (1.10x)

hours to finish the reduction. By contrast, the execution times from DD are particularly polarized. Some cases are finished in a few minutes, while in five cases, DD cannot finish the reduction in 24 hours. This is because removing consecutive tokens blindly from the variant to search for a seed is risky — it may happen to find an appropriate seed and terminate quickly, resulting in tiny differences but huge seed programs; it may also unluckily fail to find any syntactically valid seed in reasonable time. More importantly, no matter how fast or slow, DD always generates unacceptably large programs, making it infeasible to debug with.

We also measured to what extent do PPR brings time overhead than classical program reduction. Table 3.2 shows that Perses and C-Reduce average take 1.29 hours and 2.76 hours in classical program reduction, which is shorter than PPR. This is expected since Perses and C-Reduce only reduce the variant while PPR additionally reduces the seed, and the difference in between. We believe this trade-off is worthwhile since the extra minimum-sized differences provided by PPR can further ease the debugging process. In fact, PPR is even faster than C-Reduce in 10 out of 20 cases.

RQ2: On average, PPR takes 4.55 hours in reduction, which is acceptable when compared with those traditional program reduction algorithms.

3.5.4 RQ3. Effectiveness of Each Reducer in PPR

To investigate the efficacy of each reducer, we conducted an ablation study on PPR. Specifically, we created three alternatives of PPR, *i.e.*, PPR_{TD-} , PPR_{LD-} and PPR_{CO-} . Compared to PPR, each of them disables one of the reducers in Algorithm 1, *i.e.*, **MinTreeDiff** (TD), **MinListDiff** (LD) and **MinCommonality** (CO), respectively. We measured and compared their effectiveness with PPR.

Table 3.3 shows the results. Without **MinTreeDiff**, PPR_{TD-} fails to terminate in a reasonable time. This is because the difference is large ($>10,000$ tokens) and complex in most of the cases, reducing only by `ddmin` is ineffective and slow. Therefore, it is critical to reduce the difference with the guidance of syntax in **MinTreeDiff**. The size of the reduced difference of PPR_{LD-} is 3 tokens larger than PPR on average, which demonstrates the fact that **MinListDiff** serves as a complement of **MinTreeDiff** to further reduce the difference. Without **MinCommonality**, the reduction rates of seed and variant drop from 99.35%, 99.47% to 20.78%, 39.88%, respectively. The explanation for this change is that difference reducers are only responsible for reducing differences, and only **MinCommonality** is capable of removing unnecessary common code shared by seed and variant.

Table 3.3: Results when disabling a reducer or changing the order of reducers. For each scenario, we list the average final size and reduction rate. The cells with “-” indicates the reduction did not finish in 24h.

	Seed	Variant	Difference
Original	39955.80	52711.95	27880.35
PPR	259.05(99.35%)	278.05(99.47%)	24.25(99.91%)
PPR_{TD^-}	-	-	-
PPR_{LD^-}	294.90(99.26%)	311.75(99.41%)	27.20(99.90%)
PPR_{CO^-}	31653.10(20.78%)	31690.60(39.88%)	72.85(99.74%)
CO-TD-LD	791.90(98.02%)	871.55(98.35%)	423.50(98.48%)
LD-TD-CO	-	-	-

RQ3: All reducers in PPR contribute to the effectiveness of PPR, especially the **MinTreeDiff** and **MinCommonality**.

3.5.5 RQ4. The Impact of the Order of Reducers

To understand how the order of reducers impacts the reduction, we experimented with two alternative orders — reducing the commonality before reducing the difference, *i.e.*, **MinCommonality**-> **MinTreeDiff**-> **MinListDiff** (CO-TD-LD), and swapping the order of tree-based and list-based difference reduction, *i.e.*, **MinListDiff**-> **MinTreeDiff**-> **MinCommonality** (LD-TD-CO).

If **MinCommonality** is invoked ahead of all difference reducers, the average size of final seed and variant are 791.90 and 871.55, which are 2.06x and 2.13x larger than those from the original order shown in Table 3.3. The difference size is 423.50, which becomes 16x larger than the original and is almost half of the final seed/variant size on average. Further investigation explains such change: if the common code snippets instead of the difference are reduced first, then the remaining difference will take a larger proportion in the program, making tree-differencing algorithms harder to match the common part. Such impact propagates through iterations and finally produces a pair of diverged program, *i.e.*, seed and variant look very different from each other.

If **MinListDiff** is invoked before **MinTreeDiff**, PPR fails to terminate in reasonable time. Such result is in line with the behavior when **MinTreeDiff** is disabled, and demonstrates the fact that syntax guided reduction is critical in reducing large difference.

RQ4: Empirically, difference should be reduced before reducing the common parts. As for the order of difference reducers, the tree-based reducer is more effective and efficient, and thus should be given a higher priority.

3.6 Discussion

3.6.1 Usefulness of PPR in Fuzzer Design

Besides providing insights to compiler developers when debugging, PPR also offers useful feedback to those who are designing fuzzing techniques. GCC-66186 is the example showcased in the paper of Hermes [107], a compiler fuzzing tool, for illustrating the effect of their proposed mutations. The code inserted by Hermes is highlighted in blue.

To identify this bug-triggering part, the authors of Hermes reduced the variant with C-Reduce and then manually searched for patterns from the original mutation set, which involves 17 code blocks, 3,335 tokens in total. Our algorithm, in contrast, automates the procedure, finishes it in 0.85 hour, and presents the same minimized bug-triggering changes shown in Fig. 3.4.

<pre> 1 // seed.c 2 for (b = 0; b <= 7; b += 1) 3 for (d = 0; d <= 7;) { 4 5 6 7 8 if (f[b]) 9 break; 10 }</pre>		<pre> // variant.c 1 for (b = 0; b <= 7; b += 1) 2 for (d = 0; d <= 7;) { 3 if (g < 0) 4 for (; d <= 7; d += 1) 5 if (f[c]) 6 break; 7 if (f[b]) 8 break; 9 10 }</pre>
---	--	---

Figure 3.4: Final result of GCC-66186 (code snippet). Code highlighted in blue is the final differences from seed to variant.

3.6.2 Generality on Other Languages

PPR can be parameterized towards any language. To demonstrate PPR’s effectiveness on language other than C, we further conducted experiments using two prevalent programming languages, Rust and JavaScript. To build a benchmark suite similar to Benchmark-PPR, we

applied random node-level deletion, insertion and replacement on a pool of seed programs to fuzz rustc 1.58.1 and JerryScript-2.4.0 separately, and collected 9 bug-triggering programs and their corresponding seed programs for each language. Then we run PPR and DD on this benchmark suite.

As shown in Table 3.4, PPR effectively reduces the seed and variant programs by more than 90%. For variants, PPR achieves 91.86% and 94.58% reduction rates on Rust and JavaScript respectively, higher than 82.12% and 69.71% from DD. For differences, PPR reaches 4 tokens on average on both Rust and JavaScript, far lower than 57 and 24 tokens from DD. Moreover, for all cases in Rust, DD fails to find out a program for seed. Our explanation is that Rust has a strict syntactical or semantical rule for programs, so it is not likely to find a valid subset of the variant without the guidance of syntax.

Table 3.4: Results of PPR and DD on Rust and JavaScript.

Bug ID	Seed Size			Variant Size			Difference Size		
	Original	PPR	DD	Original	PPR	DD	Original	PPR	DD
RUSTC-1	157	53	0	135	50	69	22	3	69
RUSTC-2	104	40	0	100	41	78	4	1	78
RUSTC-3	552	6	0	558	8	43	6	2	43
RUSTC-4	475	23	0	497	27	44	24	4	44
RUSTC-5	184	7	0	203	16	22	22	11	22
RUSTC-6	104	51	0	99	48	46	13	3	46
RUSTC-7	83	10	0	94	14	44	11	4	44
RUSTC-8	183	11	0	182	13	134	5	2	134
RUSTC-9	992	10	0	996	16	32	22	6	32
Mean	315	23	0	318	26	57	14	4	57
Mean R		92.55%			91.86%	82.12%		72.09%	
JERRY-1	558	28	0	536	20	46	22	8	46
JERRY-2	215	11	31	191	7	32	24	4	1
JERRY-3	356	13	80	338	12	81	18	1	1
JERRY-4	93	12	91	104	15	92	11	3	1
JERRY-5	272	14	68	291	17	70	19	3	2
JERRY-6	62	10	47	80	11	49	18	1	2
JERRY-7	192	30	44	184	23	72	12	7	28
JERRY-8	199	10	40	169	11	75	36	1	35
JERRY-9	469	11	113	504	14	209	39	7	96
Mean	268	15	57	266	14	81	22	4	24
Mean R		94.25%			94.58%	69.71%		82.41%	

3.6.3 Threats to Validity

A primary threat to internal validity is that PPR relies on the tree matching algorithms to compute differences. Although tree matching algorithms such as Gumtree are widely adopted and usually effective, they may produce imperfect matchings in complex edits, particularly when changes involve large or scattered modifications. In such cases, the tree matching algorithms tend to fragment the tree structures and generate long sequences of small edits. This can negatively impact PPR’s performance, making the reduction slower or the output slightly larger.

However, this imperfection does not compromise correctness, as PPR always verifies that the reduced programs preserve the required property. Besides, we incorporate a list-based difference reduction that complements the tree-based difference reducer, further enhance the effectiveness of reduction.

The generality of our approach and results can be a threat to external validity. To mitigate this, we have evaluated PPR on 20 pairs of C programs, which reflect diversity in compilers, bug types, mutation strategies and mutation degrees. Additionally, we evaluated PPR on two other languages. The results demonstrated the effectiveness and generality of PPR on diverse programs.

3.7 Related Work

We discuss three lines of related work.

Test Input Minimization and Generalization. Delta Debugging is the foundational technique for systematically minimizing bug-triggering inputs and changes. Hierarchical Delta Debugging [86] further refines this approach by applying ddmin iteratively on different levels of a program’s parse tree from top to bottom. Perses [109, 114] enhances reduction efficiency by normalizing grammars and using transformed grammars to guide the reduction process, resulting in more effective reduction compared to HDD. Vulcan [143] further improves the performance by introducing aggressive program transformations. Specific languages also have tailored reduction tools; C-Reduce [99], designed for C/C++ programs, while J-Reduce focuses on Java bytecode [55, 56]. DDSET [41] introduces automated generalization, abstracting concrete programs into a mixture of terminal and abstract symbols to capture failure-inducing patterns. SmartCheck [96] and Extrapolate [12] specialize in generalizing Haskell programs.

All the work above mainly focus on reducing or generalizing single failure-inducing inputs and can be classified as classical program reduction algorithms. By contrast, pairwise program reduction reduces both failure-inducing input and failure-inducing changes simultaneously, in which way we believe can complement classical program reduction on compiler debugging.

Failure-Inducing Change Isolation. Delta Debugging sparked a range of related research [31, 23, 155, 21, 22] in the realm of isolating failure-inducing changes. Crisp [23] assists Java developers in locating such changes by enabling iterative selection, application, and undoing of affecting changes to pinpoint failure-inducing ones. AutoFlow [155] employs static and dynamic analysis for automated isolation of failure-inducing changes. Donaldson et al. introduced Spirv-fuzz [31], an effective reduction and deduplication approach for compiler testing, capable of reducing failure-inducing transformation sequences to smaller ones that still trigger failures. Unique in its approach, PPR not only isolates failure-inducing changes but also minimizes commonalities between two programs.

Slicing-Based Fault Localization . Technique sharing similarity with PPR is dual slicing, a slicing-based approach to locate the source of errors in software. By comparing execution traces of programs of different versions, dual slicing techniques [129, 123] can identify the differences and localize the root cause of bugs or regression faults on a program, *e.g.*, Wang *et al.* [123] studies typical programs such as grep and coreutils. However, PPR differs from program slicing *w.r.t.* the study subjects, as it focuses on programs as the input of a compiler/interpreter, instead of programs on their own. Moreover, PPR performs both reduction and isolation on a pair of program inputs, while dual slicing only isolates the root cause of errors in the program. Unlike dual slicing, which requires white-box techniques for execution traces, program reduction methods like PPR are black-box approaches, suitable for bugs within the compilers, rather than in program inputs.

3.8 Conclusion

In this chapter, we propose pairwise program reduction (PPR), a novel perspective to complement classical program reduction techniques. Unlike Perses and C-Reduce, PPR takes as input a pair of programs. It reduces not only the seed and variant programs, but also the differences between them. To evaluate PPR, we construct Benchmark-PPR containing programs that trigger bugs in GCC and LLVM. The experiment conducted with Benchmark-PPR shows that final programs from PPR are several orders of magnitude smaller than those from DD, and even comparable to classical program reduction *w.r.t.* size.

PPR provides more insightful difference than DD for both compiler developers and fuzzing designers. Finally, evaluation on Rust and JavaScript demonstrates PPR’s generality on other languages.

Chapter 4

Toward a Better Understanding of Probabilistic Delta Debugging

This chapter conducts a theoretical analysis of ProbDD, a state-of-the-art reduction algorithm, to deepen the understanding of the factors underlying its superior performance. Based on this analysis, we simplify the algorithm and propose CDD, which performs on par with ProbDD but is easier to comprehend and implement. Finally, we offer actionable suggestions to guide researchers and users in effectively applying ddmin and its variants in practice.

4.1 Introduction

Delta Debugging [145] is a seminal family of algorithms designed for software debugging, among which ddmin stands out as a classic test input minimization (*a.k.a.*, test input reduction) algorithm. Given a list L of elements (modeling the test input) and a property ψ that L exhibits, ddmin aims to remove elements in L that are irrelevant to ψ , such that the resulting list is smaller than L yet still satisfies ψ . The ddmin algorithm plays a crucial role in software testing, debugging and maintenance [37, 82, 128, 10, 15], since compact and informative bug-triggering inputs are easier for developers to effectively identify root causes than large bug-triggering inputs with bug-irrelevant information [28, 127, 116, 107].

To minimize a test input I that satisfies ψ , ddmin has been used in two primary manners. In the first manner, I is initially segmented into a list, denoted as L , which could be segmented based on characters, tokens, lines, *etc.* Subsequently, ddmin is directly

applied to L [145, 31]. Alternatively, `ddmin` serves as a pivotal component within advanced, structure-aware test input minimization algorithms, including `Perses` [109], `HDD` [86], `C-Reduce` [99], and `Chisel` [45]. These algorithms leverage the inherent structures of I to expedite the minimization process or further reduce its size. Generally, these algorithms initiate by parsing I into a tree structure, such as a parse tree. They then iteratively extract a list L of tree nodes from the tree using heuristics and apply `ddmin` to L to gradually condense the tree. Both manners underscore the fundamental role of `ddmin` as the cornerstone of test input minimization.

In the past years, different variants of `ddmin` have been proposed to improve its performance [45, 119, 39, 49, 158], among which Probabilistic Delta Debugging (`ProbDD`) [119] is the state of the art, with notable superiority to other algorithms [145, 45]. When reducing L , `ProbDD` utilizes a theoretical probabilistic model based on Bayesian optimization to predict how likely every element in L is essential to preserve the property ψ , by assigning a probability to each element. `ProbDD` prioritizes deleting elements with lower probabilities, as such elements generally have a lower possibility of being ψ -relevant. Before each deletion attempt, an optimal subset of elements is determined by maximizing the Expected Reduction Gain, which is defined as the expected number of elements removed in each attempt. Higher Expected Reduction Gain is preferred, as it indicates an expectation to delete more elements through this attempt. If the deletion of this subset fails to preserve ψ , the probabilistic model increases the probability assigned to each element in the subset. As reported [120], aided by such a probabilistic model, `ProbDD` significantly outperforms `ddmin` by reducing the execution time and the query number (a query is a run of the property test ψ).

However, this probabilistic model in `ProbDD` is rather intricate, and the underlying mechanisms for its superior performance have not been adequately studied. The original paper of `ProbDD` merely showed its performance numbers without deep ablation analysis on such achievements. Specifically, the following questions are important to the research field of test input minimization, but have not been answered yet.

1. What role do probabilities play in `ProbDD`, and can they be simplified without impacting performance?
2. What specific bottlenecks does `ProbDD` overcome to achieve improvement compared to `ddmin`?
3. How does randomness in `ProbDD` contribute to the performance improvement?
4. What are the potential limitations of `ProbDD`?

Gaining a deeper understanding of the state of the art, *i.e.*, ProbDD, is highly valuable for test input minimization tasks. By clarifying the intrinsic reasons behind its superiority, we can facilitate researchers to understand the essence of the probabilistic model, as well as its strengths and limitations. Such demystification, in our view, paves the way for enlightening future research and guides users to more effectively apply ddmin and its variants for test input minimization.

To this end, we conduct the first in-depth analysis of ProbDD, starting by theoretically simplifying its probabilistic model. In the original ProbDD, probabilities are used to calculate the Expected Reduction Gain, which is subsequently used to determine the next subset size. However, this process necessitates iterative calculations, impeding the simplification and comprehension of ProbDD. In our study, we initially establish the analytical correlation between the probability and subset size, allowing for probabilities and subset sizes to be explicitly calculated through formulas, thus eliminating the need for iterative updates. Further, through mathematical derivation, we discover that the probability and subset size can be considered nearly independent, each varying at an approximate ratio on their own. By theoretical prediction, the probability increases approximately by a factor of $\frac{1}{1-e^{-1}}$ (≈ 1.582), and the subset size can be deduced by this probability, thus providing the potential for simplifying ProbDD.

Building upon our theoretical analysis, we conducted extensive evaluations of ddmin, ProbDD, and CDD across 76 diverse benchmarks. The experimental results confirm the correctness of our theoretical analysis, demonstrating how ProbDD addresses bottlenecks in ddmin by skipping inefficient queries, reveals the impact of randomness on results, and highlights the limitations of ProbDD. These findings provide valuable guidance for future research and the development of test input minimization algorithms.

Based on the aforementioned analysis, we propose **Counter-Based Delta Debugging** (CDD), a simplified version of ProbDD, to explain ProbDD’s high performance. By replacing probabilities with counters, CDD eliminates the probability computations required by ProbDD, thus reducing theoretical and implementation complexity. Our experiments demonstrate that CDD aligns with ProbDD in both effectiveness and efficiency, which validates our previous analysis and findings.

Key Findings. Through both theoretical analysis and empirical experiments, our key findings are:

1. Through theoretical derivation, the probabilities in ProbDD essentially serve as monotonically increasing counters, and can be simplified. This suggests that the probability mechanism itself may not be a critical factor in ProbDD’s superior performance.

2. The performance bottlenecks addressed by ProbDD are inefficient deletion attempts on complements of subsets and previously tried subsets, which should be considered to enhance efficiency.
3. Randomness in ProbDD has no significant impact on the performance. Test input minimization is an NP-complete problem, and randomness in ProbDD does not produce smaller results.
4. ProbDD is faster than ddmin, but at the cost of not guaranteeing 1-minimality (A list is considered to have 1-minimality if removing any single element from it results in the loss of its property). The trade-off between effectiveness and efficiency is inevitable, and should be leveraged accordingly in different scenarios.

Contributions. We make the following major contributions.

- We perform the first in-depth theoretical analysis for ProbDD, the state-of-the-art algorithm in test input minimization tasks, and identify the latent correlation between the subset size and the probability of elements.
- We propose CDD, a much simplified version of ProbDD.
- We evaluate ddmin, ProbDD and CDD on 76 benchmarks, validating the correctness of our theoretical analysis. Additional experiments and statistical analysis on ProbDD further explain its superior performance, reveal the effectiveness of randomness, and demonstrate the limitations of ProbDD.
- To enable future research on test input minimization, we release the artifact publicly for replication[149]. Additionally, we have integrated CDD into the Perses project, available at <https://github.com/uw-pluverse/perses>.

Chapter Organization. The remainder of the chapter is structured as follows: § 4.2 introduces the symbols used in this study and the detailed workflow of ddmin and ProbDD. § 4.3 presents our in-depth analysis on ProbDD, simplifying the model of probability and subset size. § 4.4 describes empirical experiments and results, from which additional findings are derived. § 4.5 introduces CDD, which simplifies ProbDD based on our earlier findings while maintaining equivalent performance. § 4.7 illustrates related work and § 4.8 concludes this study.

4.2 Preliminaries

To facilitate comprehension, Table 4.1 lists all the important symbols used in this chapter. Next, this section introduces ddmin and ProbDD, with the running example shown in Fig. 4.1.

Table 4.1: The symbols used in this chapter.

Symbol	Description	Symbol	Description
L	the list to minimize	s	the size of S
ψ	the property to preserve	$E(s)$	Expected Reduction Gain with the first s elements
l_i	the i -th element of L	e	Euler’s number
$l_i.p$	the probability of l_i	r	the round number
v_i	a variant of L	s_r	the subset size in round r
S	a subset of L	p_r	the probability of each element in round r

```
l1:import math, sys
l2:input = sys.argv[1]
l3:a = int(input)
l4:b = math.e
l5:c = 3
l6:d = pow(b, a)
l7:c = math.log(d, b)
l8:crash(c)
```

(a) Original.

```
l1:import math, sys
l2:input = sys.argv[1]
l3:a = int(input)
l4:b = math.e
l5:c = 3
l6:d = pow(b, a)
l7:c = math.log(d, b)
l8:crash(c)
```

(b) By ddmin.

```
l1:import math, sys
l2:input = sys.argv[1]
l3:a = int(input)
l4:b = math.e
l5:c = 3
l6:d = pow(b, a)
l7:c = math.log(d, b)
l8:crash(c)
```

(c) By ProbDD.

Figure 4.1: A running example in Python. Fig. 4.1(a) shows the original program, represented as a list of 8 elements ($l_1, l_2, \dots, l_8$), in which l_8 (*i.e.*, **crash(c)**) triggers the crash. Fig. 4.1(b) and Fig. 4.1(c) show the minimized results by ddmin and ProbDD, with removed elements masked in gray. Both minimized programs still trigger the crash. Note that ProbDD cannot consistently guarantee the result in Fig. 4.1(c) and might produce larger results, due to its inherent randomness.

4.2.1 The ddmin Algorithm

The ddmin algorithm [145] is the first algorithm to systematically minimize a bug-triggering input to its essence, which has been widely adopted in program reduction[109, 86, 99],

Table 4.2: Step-by-step outcomes from ddmin on the running example. In each column, a variant is generated and tested against the property ψ . These variants are sequentially generated from left to right. The first row displays the variant identifier, and the second row displays round number r and subset size s . In the following rows, the symbol “✓” denotes an element is included by a certain variant, while **gray** cells signify that the element have been removed. For the last row, **T** indicates that the variant still preserves the property ψ , whereas **F** indicates not.

Initial	Variables	v ₁	v ₂	v ₃	v ₄	v ₅	v ₆	v ₇	v ₈	v ₉	v ₁₀	v ₁₁	v ₁₂	v ₁₃	v ₁₄	v ₁₅	v ₁₆	v ₁₇	v ₁₈	v ₁₉	v ₂₀	v ₂₁	v ₂₂	v ₂₃	v ₂₄	v ₂₅	v ₂₆	v ₂₇	v ₂₈	v ₂₉	v ₃₀			
Element	Round	r = 1 (s=4)				r = 2 (s=2)						r = 3 (s=1)																						
<i>l</i> ₁		✓		✓					✓	✓	✓	✓										✓	✓	✓	✓	✓		✓	✓	✓	✓	✓		
<i>l</i> ₂		✓		✓					✓	✓	✓	✓	✓									✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		
<i>l</i> ₃		✓			✓				✓	✓	✓	✓			✓							✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		
<i>l</i> ₄		✓			✓				✓	✓	✓	✓			✓							✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		
<i>l</i> ₅			✓			✓			✓	✓	✓	✓				✓						✓	✓	✓	✓								✓	✓
<i>l</i> ₆			✓				✓		✓	✓	✓	✓					✓					✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
<i>l</i> ₇			✓				✓		✓	✓	✓	✓						✓				✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
<i>l</i> ₈			✓					✓	✓	✓	✓	✓							✓			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
ψ		F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	T	F	F	F	F	F	F	F		

software debloating[24, 45] and test suites reduction[42, 43]. It takes the following two inputs:

- L : a list of elements representing a bug-triggering input. For example, L can be a list of bytes, characters, lines, tokens, or parse tree nodes extracted from the bug-triggering input.
- ψ : a property that L has. Formally, ψ can be defined as a predicate that returns **T** if a list of elements preserves the property, **F** otherwise.

and returns a minimal subset of L that still preserves ψ , from which excluding any single element will make the minimal subset lose ψ . This algorithm has been widely used in practice to facilitate developers in debugging [99, 109, 31, 59]. It generally consists of the following three steps.

Initialize. Start by setting the initial subset size s to half of the input list L , *i.e.*, $s=|L|/2$.

Step 1: Minimize to Subset. Partition L into subsets of size s . For each subset S , check whether S alone satisfies ψ . If yes, keep only S and restart from Step 1 with $L = S$ and the subset size as half of the new L ; otherwise, go to Step 2.

Step 2: Minimize to Complement. Partition L into subsets of size s . For each subset S , check whether the complement of S (*i.e.*, $L/S = \{e | e \in L \wedge e \notin S\}$) satisfies ψ . If yes,

keep the complement of S and restart from Step 2 with $L = L/S$; otherwise, go to Step 3.

Step 3: Subdivide. If any of the remaining subsets has at least two elements and thus can be further divided, halve the subset size, *i.e.*, $s = s/2$ and go back to Step 1. If no subset can be further divided (*i.e.*, the subset size is 1), ddmin terminates and returns the remaining elements as the result.

Round Number r . Note that we introduce a round number r at the second column of Table 4.2. Within each round, the list L is divided into subsets of a fixed size, on which Step 1 and Step 2 are applied. A new round begins when no further progress can be made with the current subset size. This round number is *not explicitly* present in the original ddmin algorithm but exists *implicitly*. In subsequent sections, we will also use this concept to introduce and simplify the ProbDD algorithm.

Table 4.2 illustrates the step-by-step minimization process of ddmin with the running example in Fig. 4.1. Initially, the input L is $[l_1, l_2, \dots, l_8]$. The ddmin algorithm iteratively generates variants by gradually decreasing the subset size from 4 to 1.

1. Round 1 ($s=4$). At the beginning, ddmin splits L into two subsets and generates two variants v_1 and v_2 . However, neither of them preserves ψ .
2. Round 2 ($s=2$). Next, ddmin continues to subdivide these two subsets into smaller ones, and generates eight variants (*i.e.*, $v_3, v_4, \dots, v_{10}$) by using these subsets and their complements. Specifically, the first four variants (v_3, v_4, v_5, v_6) are the subsets, and the next four variants (v_7, v_8, v_9, v_{10}) are the complements of these subsets. Again, none of these eight variants preserves ψ .
3. Round 3 ($s=1$). Finally, ddmin decreases subset size s from 2 to 1, and generates more variants. This time, v_{23} , which is the complement of the subset $\{l_5\}$, preserves ψ . Hence, the subset $\{l_5\}$ is permanently removed from L . Then for each of the remaining subsets $\{l_1\}, \{l_2\}, \dots, \{l_8\}$, ddmin restarts testing the complement of each subset, *i.e.*, from v_{24} to v_{30} . However, none of these variants preserves ψ , and no subset can be further divided, so ddmin terminates with the variant v_{23} as the final result.

4.2.2 Probabilistic Delta Debugging (ProbDD)

Wang *et al.* [119] proposed the state-of-the-art algorithm ProbDD, significantly surpassing ddmin in minimizing bug-triggering programs on C compilers and benchmarks in software debloating. ProbDD employs Bayesian optimization [95] to model the minimization problem.

ProbDD assigns a probability to each element in L , representing its likelihood of being essential for preserving the property ψ . At each step during the minimization process, ProbDD selects a subset of elements expected to yield the highest Expected Reduction Gain, and targets these elements in the subset for deletion. In this section, we outline ProbDD's workflow in Algorithm 5, paving the way for a deeper understanding and analysis of ProbDD.

Initialize (line 1). In L , ProbDD assigns each element an initial probability p_0 on line 1, representing the prior likelihood that each element cannot be removed.

Step 1: Select Elements (line 4, line 14–line 24). First, ProbDD sorts the elements in L by probability in ascending order on line 14, and the order of elements with the same probability is determined randomly. Then, on line 19, it calculates the subset to be removed in the next attempt via the proposed Expected Reduction Gain $E(s)$, as shown in Equation (4.1), with $E(s)$ denoting the expected gain obtained via removing the *first s elements* in L selected for deletion, and $l_i.p$ denoting the current probability of the i -th element in L .

$$E(s) = s \times \prod_{i=1}^s (1 - l_i.p) \quad (4.1)$$

Note that ProbDD has an invariant that the subset S chosen for deletion attempt is always the first s elements in L . Every time, the first s^* elements are selected as the optimal subset S , where s^* maximizes the Expected Reduction Gain $E(s)$, elaborated as Equation (4.2).

$$s^* = \arg \max_s E(s) \quad (4.2)$$

Step 2: Delete the Subset (line 5–line 9). If ψ is still preserved after the removal of S , ProbDD removes subset S on line 6, *i.e.*, keeps only the complement of S , and proceeds to Step 1. If ψ cannot be preserved after the removal, on lines 8 and 9, ProbDD updates the probability of each element in the subset S via Equation (4.3), and resumes at Step 1. It is important to note that if an element l_i has been individually deleted but failed, its probability $l_i.p$ will be set to 1, indicating that this element cannot be removed and will no longer be considered for deletion.

$$l_i.p \leftarrow \frac{l_i.p}{1 - \prod_{l \in S} (1 - l.p)} \quad (4.3)$$

Step 3: Check Termination (line 3). If every element either has been deleted, or

Algorithm 5: ProbDD(L, ψ)

Input: L : a list to be minimized.
Input: $\psi : \mathbb{P} \rightarrow \mathbb{B}$: the property to be preserved by L .
Input: p_0 : the initial probability given by the user.
Output: the minimized list that still exhibits the property ψ .
// Initialize the probability of each element with p_0

```
1 foreach  $l \in L$  do  $l.p \leftarrow p_0$ 
  /* The round number  $r$ , initially 0.  $r$  is not explicitly used in the original
    ProbDD algorithm, but is shown to illustrate ProbDD's implicit principles. */
2  $r \leftarrow 0$ 
3 while  $\exists l \in L : l.p < 1$  do
  // Select elements from  $L$  for deletion attempt.
4   $S \leftarrow \text{SelectSubset}(L)$ 
  // Check if removing the subset preserves the property
5   $\text{temp} \leftarrow L \setminus S$ 
6  if  $\psi(\text{temp}) = \text{T}$  then  $L \leftarrow \text{temp}$ 
7  else
  // Calculate the factor to update probabilities
8   $\text{factor} \leftarrow \frac{1}{1 - \prod_{l \in S} (1 - l.p)}$ 
  // Update the probabilities of elements in the subset
9  foreach  $l \in S$  do  $l.p \leftarrow \text{factor} \times l.p$ 
10 if All elements' probability have been updated then
  // Move to the next round.
11   $r = r + 1$ 
12 return  $L$ 
```

Function SelectSubset(L):
Input: L : a list of elements to be reduced.
Output: The subset of elements that maximizes the Expected Reduction Gain.
/* Sort elements in L by ascending probability. Shuffle elements with the
same probability. */

```
14  $\text{sortedL} \leftarrow \text{RandomizeThenSort}(L)$ 
15  $S \leftarrow \emptyset$ 
16  $\text{currentMaxGain} \leftarrow 0$ 
17 foreach  $l \in \text{sortedL}$  do
18    $\text{tempSubset} \leftarrow S \cup \{l\}$ 
19    $\text{gain} \leftarrow |\text{tempSubset}| \times \prod_{l \in \text{tempSubset}} (1 - l.p)$ 
20   if  $\text{gain} > \text{currentMaxGain}$  then
21      $\text{currentMaxGain} \leftarrow \text{gain}$ 
22      $S \leftarrow \text{tempSubset}$ 
23   else break
24 return  $S$ 
```

possesses a probability of 1, ProbDD terminates. If not, it returns to Step 1.

Round Number r . Similar to the concept of rounds in ddmin (see Table 4.2), ProbDD also has an *implicit* round number r , as introduced on line 2 in Algorithm 5 and the second row of Table 4.3. During a round, the subset size is the same and every subset in L is attempted for deletion. Once the probabilities of all elements have been updated, the next round begins (*i.e.*, $r \leftarrow r + 1$ on line 11).

Table 4.3: Step-by-step outcomes from ProbDD on the running example. Similar to Table 4.2, round number, subset size and the details of each variants are presented. For each variant, the probability of each element is noted alongside.

Initial		Variants	v ₁	v ₂		v ₃		v ₄		v ₅		v ₆		v ₇		v ₈
Element	Prob	Round	r = 1 (s=4)			r = 2 (s=2)			r = 3 (s=1)							
<i>l</i> ₁	0.25		0.37	✓	0.37	0.61	✓	0.61	✓	0.61	✓	0.61				
<i>l</i> ₂	0.25		✓	0.25												
<i>l</i> ₃	0.25		✓	0.25												
<i>l</i> ₄	0.25		0.37	✓	0.37	✓	0.37	0.61								
<i>l</i> ₅	0.25		0.37	✓	0.37	0.61	✓	0.61	✓	0.61		1	✓	1	✓	1
<i>l</i> ₆	0.25		✓	0.25												
<i>l</i> ₇	0.25		✓	0.25												
<i>l</i> ₈	0.25		0.37	✓	0.37	✓	0.37	0.61	✓	0.61	✓	0.61	✓	0.61	1	
<i>ψ</i>			F	T		F	F		T	F		T		F		

Table 4.3 illustrates the step-by-step results of ProbDD. Following the study of ProbDD [119], the initial probability p_0 is set to 0.25, resulting in subsets with a size of 4 as per Equation (4.2).

1. Round 1 ($s=4$). Similar to the example in the original paper of ProbDD [119], we assume ProbDD selects (l_1, l_4, l_5, l_8) to delete due to the randomness, thus resulting in the variant v_1 . However, v_1 fails to exhibit ψ , leading to the probability of these selected elements being updated from 0.25 to $\frac{0.25}{1-(1-0.25)^4} \approx 0.37$, based on Equation (4.3). Next, the remaining elements with lower probability, *i.e.*, (l_2, l_3, l_6, l_7) , are prioritized and selected for deletion, resulting in v_2 . This time, the property test passes and these elements are removed.
2. Round 2 ($s=2$). Given that all probabilities of remaining elements become 0.37, the next subset size becomes 2. Subsequently, subset (l_1, l_5) are attempted to remove in v_3 and later subset (l_4, l_8) are attempted to remove in v_4 , though no subset can be successfully removed. After these two attempts, all probabilities update to $\frac{0.37}{1-(1-0.37)^2} \approx 0.61$.

3. Round 3 ($s=1$). Finally, the subset size becomes 1, so each individual element is selected to remove alone. The elements l_4 and l_1 are finally removed from the final result in v_5 and v_7 , respectively, while l_5 and l_8 are verified as non-removable, thus being returned as the final result.

4.3 Delving Deeper into Probability and Size

Beginning with this section, we will systematically present our findings. Each finding will be introduced by first stating the result, followed by the explanation. In this section, we theoretically analyze the trend of probability changes across rounds, and the approach to derive the optimal subset size.

4.3.1 On the Probability in ProbDD

Finding 1: The probability assigned to each element increases monotonically with the round number r , by a factor of approximately 1.582. Essentially, the probability for each element can be expressed as a function of r and p_0 , i.e.,

$$p_r \approx 1.582^r \times p_0$$

An Illustrative Example. The running example illustrated in Table 4.3 leads to this finding. Observation reveals that after each element has been attempted for deletion once, i.e., completing one round, the probabilities of all remaining elements are updated. The initial probability is 0.25; after v_2 , it changes to 0.37; following v_4 , it increases to 0.61; and by the end of v_8 , it reaches 1. Consequently, we hypothesize that with each deletion attempt, the probability approximately increases in a predictable manner. Through appropriate simplification, we can theoretically model this trend, and thereby model the entire progression of probability changes.

Assumption for Theoretical Analysis

Besides the above observation from a concrete example, theoretical analysis is necessary. To refine the mathematical model of ProbDD for easier representation, analysis and derivation, we assume that the number of elements in L is always divisible by the subset size. With this assumption, the probability of each element will be updated in the same manner; as a result, before and after each round, the probabilities of all elements are always the same, as

shown in lemma 4.3.1. This assumption is often applicable in practice. For instance, in the running example in Table 4.3, before each round, the probabilities associated with each remaining element are identical, ensuring that all subsets are of identical size. Furthermore, the probabilities of elements are updated to the same next value after the round.

Lemma 4.3.1. *If the number of elements in L is always divisible by the subset size, then after each round, the probabilities of all elements will always remain the same.*

Proof. We use mathematical induction to prove this lemma.

Base Case. Initially, all probabilities are set to the same value. Hence, before the first round, the probabilities of all elements are identical.

Inductive Step. Assume that before a given round, the probabilities of all elements are identical (induction hypothesis). After failing to delete a subset S , ProbDD updates the probability of each element of S according to Equation (4.3). This formula depends solely on two factors: the current probability of each element of S , *i.e.*, $l_i.p$, and the size of the subset $|S|$. For $l_i.p$, by the induction hypothesis, all elements have the same probability at the beginning of the round; for $|S|$, if the total number of elements in L is divisible by the subset size, then every subset in the round will have the same size $|S|$. Therefore, both factors $l_i.p$ and $|S|$ are identical for all elements in a subset, and the probabilities of these elements are updated to the same new value using Equation (4.3). Furthermore, as all subsets undergo the same update process, the probabilities of all elements in the list will remain identical at the end of the round.

Conclusion. The probabilities of all elements remain identical at the end of this round. $\square$

Consequently, as long as the total number of elements is always divisible by the subset size, the probabilities of all elements will remain identical throughout the process. Take the running example in Table 4.3 as a demonstration. During the reduction, the number of elements is always divisible by the subset size in each round, *i.e.*, $s=4$, $s=2$, $s=1$. Therefore, starting with an initial probability of 0.25, the probability of each elements remain identical after each round, being 0.37, 0.61 and 1, respectively.

While it is not always possible for the number of elements to be divisible by the subset size, the elements will still be partitioned as evenly as possible. However, such indivisibilities make the theoretical simplification of ProbDD nearly impossible. Based on our observation when running ProbDD, being slightly uneven during partitioning does not significantly affect probability updates. Moreover, we will demonstrate that the simplified algorithm derived from this assumption has no significant difference from ProbDD in § 4.5, via thorough experimental evaluation.

Probability *v.s.* Subset Size Correlation

In the second step, we derive the correlation between probability and subset size. Based on the assumption in the previous step, the probability of each element is identical and represented as p_r in round r , thus the formula of Expected Reduction Gain from Equation (4.1) can be simplified to

$$E(s) = s \times (1 - p_r)^s \quad (4.4)$$

Given the probability of elements p_r in the round r , s_r can be derived through gradient-based optimization, *i.e.*, $E'(s_r) = 0$. Therefore, the optimal size s_r to maximize $E(s)$ is $-\frac{1}{\ln(1-p_r)}$. Subsequently, we can also deduce the next probability to be $p_{r+1} = \frac{p_r}{1-(1-p_r)^{s_r}}$. In summary, the correlation between probability and subset size can be simplified as Equation (4.5) and Equation (4.6), in which subset size s_r is determined by probability p_r , and probability p_{r+1} in the next round is determined by both p_r and s_r .

$$\begin{cases} s_r = -\frac{1}{\ln(1-p_r)} \\ p_{r+1} = \frac{p_r}{1-(1-p_r)^{s_r}} \end{cases} \quad (4.5)$$

$$(4.6)$$

Trend of Probability Changes

Through Equation (4.6), $p_{r+1} > p_r$ always holds, indicating a monotonic increase of the probability of elements. However, there is still room for simplification, as s_r can be represented by p_r , implying that p_{r+1} can be represented solely by p_r .

Lemma 4.3.2. p is increased by a factor $\frac{1}{1-e^{-1}}$, *i.e.*,

$$p_r = \frac{p_{r-1}}{1-e^{-1}} = \frac{p_0}{(1-e^{-1})^r} \quad (4.7)$$

Proof. Given $s_r = -\frac{1}{\ln(1-p_r)}$, we can deduce that $1 - p_r = e^{-\frac{1}{s_r}}$.

Subsequently, we substitute $1 - p_r$ into Equation (4.6), obtaining

$$\begin{aligned} p_{r+1} &= \frac{p_r}{1-(1-p_r)^{s_r}} = \frac{p_r}{1-(e^{-\frac{1}{s_r}})^{s_r}} \\ &= \frac{p_r}{1-e^{-1}} \approx 1.582 \times p_r \end{aligned}$$

Equivalently, the approximate probability after round r can be derived given only p_0 , *i.e.*,

$$p_r = \frac{p_0}{(1 - e^{-1})^r} \approx 1.582^r \times p_0$$

□

Therefore, through empirical observations on the running example, coupled with theoretical derivation and simplification, we have identified the pattern of probability changes *w.r.t.* the round number r , *i.e.*, $p_r = \frac{p_0}{(1 - e^{-1})^r} \approx 1.582^r \times p_0$.

4.3.2 On the Size of Subsets in ProbDD

Finding 2: The size of subsets in r -th round can be analytically pre-determined given only the probability of this round, *i.e.*, $s_r = \arg \max_{s \in \mathbb{N}^+} s \times (1 - p_r)^s$, which is either $\lfloor -\frac{1}{\ln(1-p_r)} \rfloor$ or $\lceil -\frac{1}{\ln(1-p_r)} \rceil$.

Based on the Finding 1, the probability p_r can be approximately estimated by the current round number r via a factor. Consequently, we can further derive the subset size s_r by maximizing the Expected Reduction Gain in ProbDD.

Lemma 4.3.3. *The optimal subset size s_r in round r is either $\lfloor -\frac{1}{\ln(1-p_r)} \rfloor$ or $\lceil -\frac{1}{\ln(1-p_r)} \rceil$.*

Proof. The Expected Reduction Gain is determined by the formula $E(s_r) = s_r \times (1 - p_r)^{s_r}$, which increases initially with s_r and then decreases as s_r grows further, enabling the optimal solution to be identified through derivative analysis. Therefore, we can deduce the optimal s_r by solving $E'(s_r) = 0$. Therefore, the optimal size of subsets s_r in r -th round is $-\frac{1}{\ln(1-p_r)}$, which will be rounded to either $\lfloor -\frac{1}{\ln(1-p_r)} \rfloor$ or $\lceil -\frac{1}{\ln(1-p_r)} \rceil$. The final subset size should be chosen based on which integer results in a larger Expected Reduction Gain. □

lemma 4.3.3 allows the subset size to be analytically pre-determined, thus providing the potential for simplification of ProbDD and leading to the proposal of CDD (detailed in § 4.5).

4.4 Empirical Experiments

In addition to the theoretical derivation above, we conduct an extensive experimental evaluation on ddmin and ProbDD to gain deeper insights and achieve further discoveries.

Specifically, we reproduce the experiments on `ddmin` and `ProbDD` by Wang *et al.* [119], and then delve deeper into `ProbDD`, analyzing its randomness, the bottlenecks it overcomes, and its 1-minimality. Furthermore, we evaluate our proposed CDD (which will be presented in § 4.5), validating our previous theoretical analysis. Due to limited space, we present the results of both `ProbDD` and CDD together within this section, but this section primarily focuses on discussing `ProbDD`, while the next section will focus on CDD.

4.4.1 Benchmarks

To extensively evaluate `ddmin`, `ProbDD` and CDD, we use the following three benchmark suites (76 benchmarks in total), covering various use scenarios of minimization algorithms.

- **BM_C**: 20 large bug-triggering programs in C language, each of which triggers a real-world compiler bug in either LLVM or GCC. The original size of benchmarks ranges from 4,397 tokens to 212,259 tokens. This benchmark suite has been used to evaluate test input minimization work [109, 119, 142, 141].
- **BM_{DBT}**: source programs of 10 command-line utilities. The original size of benchmarks ranges from 34,801 tokens to 163,296 tokens. This benchmark suite was collected by Heo *et al.* [45] and used to evaluate software debloating techniques [45, 97, 6].
- **BM_{XML}**: 46 XML inputs triggering 8 unique bugs in Basex, a widely-used XML processing tool. The original size of benchmarks ranges from 19,290 tokens to 20,750 tokens. This benchmark suite is generated via Xpress [73] and collected by the authors of this study, as the original XML dataset used in `ProbDD` paper is not publicly available.

4.4.2 Evaluation Metrics

We measure the following aspects as metrics.

Final Size. This metric assesses the effectiveness of reduction. When reducing a list L with a certain property ψ , a smaller final list is preferred, indicating that more irrelevant elements have been successfully eliminated. In all benchmark suites, the metric is measured by *the number of tokens*.

Execution Time. The execution time of a minimization algorithm reflects its efficiency. A minimization algorithm taking less time is more desirable, and execution time is measured

in *seconds*.

Query Number. This metric further evaluates the algorithm’s efficiency. During the reduction process, each time a variant is produced, the algorithm verifies whether this variant still preserves the property ψ , referred to as a query. Since queries consume time, a lower query number is favorable.

P-value. We calculate the p-value via Wilcoxon signed-rank test[134] between every two algorithms, to investigate whether the performance differences are significant. A p-value below 0.05 indicates that we can reject the null hypothesis (which assumes no difference in performance) at the 0.05 significance level. Otherwise, we fail to reject the null hypothesis, suggesting that the observed difference may not be statistically significant.

4.4.3 The Wrapping Frameworks

The ddmin algorithm and its variants usually serve as the fundamental algorithm. To apply them to a concrete scenario, an outer wrapping framework is generally needed to handle the structure of the input. In our evaluation, we choose the same wrapping frameworks as those used by ProbDD paper. For those tree-structured bug-triggering inputs, *i.e.*, BM_C and BM_{XML} , we use Picireny 21.8 [60], an implementation of HDD [86]. Picireny parses such inputs into trees, and then invokes Picire 21.8 [59], an open-sourced Delta Debugging library with ddmin, ProbDD and CDD implemented, to reduce each level of the trees. For software debloating on BM_{DBT} , Chisel [45] is employed, in which ddmin, ProbDD and CDD are integrated.

All experiments are conducted on a server running Ubuntu 22.04.3 LTS, with 4 TB RAM and two Intel Xeon Gold 6348 CPUs @ 2.60GHz. To ensure the reproducibility, we employ docker images to release the source code and the configuration. Each benchmark is reduced using a single thread. Following the ProbDD paper, we run each algorithm on each benchmark 5 times and calculate the geometric average results.

4.4.4 Reproduction Study of ProbDD

To comprehensively reproduce the results of ProbDD [119], we evaluate ddmin and ProbDD using three benchmark suites, containing a total of 76 benchmarks. Following the settings of ProbDD [119], we set the empirically estimated remaining rate as the initialization

Table 4.4: The final size, execution time and query number of ddmin, ProbDD and CDD on BM_C and BM_{DBT}. To address significant variations across benchmarks, the geometric mean rather than the arithmetic mean is employed, providing a smoother measure of the average.

	Benchmark	Original size (#)	Final size (#)			Execution time (s)			Query number		
			ddmin	ProbDD	CDD	ddmin	ProbDD	CDD	ddmin	ProbDD	CDD
BM _C	LLVM-22382	9,987	350	353	350	1,917	1,163	1,005	11,388	5,973	5,262
	LLVM-22704	184,444	786	764	745	27,924	12,418	11,371	52,412	15,425	14,025
	LLVM-23309	33,310	1,316	1,338	1,265	17,619	9,991	10,828	55,968	19,195	17,953
	LLVM-23353	30,196	321	336	324	3,117	1,874	1,400	11,719	5,757	4,492
	LLVM-25900	78,960	941	932	937	7,258	3,683	3,104	35,740	12,553	12,817
	LLVM-26760	209,577	520	503	498	13,123	5,876	5,210	30,063	9,261	9,792
	LLVM-27137	174,538	972	1,040	966	63,971	22,208	23,154	122,516	22,292	20,460
	LLVM-27747	173,840	431	463	510	6,545	4,238	2,932	20,000	8,193	5,992
	LLVM-31259	48,799	1,033	965	1,035	13,815	7,497	8,205	35,135	10,776	13,445
	GCC-59903	57,581	1,185	845	743	9,067	4,879	3,587	47,698	15,725	13,844
	GCC-60116	75,224	1,615	1,628	1,617	44,287	27,202	27,195	80,059	27,268	23,204
	GCC-61383	32,449	959	966	974	12,579	6,514	6,566	43,716	13,593	14,149
	GCC-61917	85,359	882	908	884	6,740	3,591	2,953	31,414	12,908	14,194
	GCC-64990	148,931	744	876	681	21,633	11,890	11,119	44,521	16,074	12,112
	GCC-65383	43,942	706	701	709	5,132	3,543	3,358	25,051	8,591	9,686
	GCC-66186	47,481	1,012	981	1,001	14,280	7,236	12,478	47,253	12,741	19,094
	GCC-66375	65,488	1,128	1,141	1,204	23,576	14,182	23,229	47,339	15,690	16,469
	GCC-70127	154,816	934	973	930	36,390	23,925	24,143	54,925	15,388	15,219
	GCC-70586	212,259	1,583	1,561	1,572	28,859	13,519	15,818	102,603	24,716	30,715
	GCC-71626	4,397	184	184	184	119	114	99	1,608	1,156	1,220
	Mean	64,599	777	775	760	10,486	5,828	5,676	34,013	11,652	11,512
BM _{DBT}	bzip2-1.0.5	70,530	20,710	20,747	20,756	137,463	105,782	92,058	54,034	23,240	19,846
	chown-8.2	43,869	9,087	9,303	9,310	38,902	25,616	7,625	50,487	8,278	8,208
	date-8.21	53,442	20,604	20,738	21,120	115,292	16,486	15,378	139,934	14,235	13,928
	grep-2.19	127,681	28,723	28,627	28,990	97,821	85,694	93,552	277,130	42,246	37,607
	gzip-1.2.4	45,929	17,065	17,068	17,077	73,403	55,520	75,913	147,035	27,569	61,032
	mkdir-5.2.1	34,801	8,625	8,782	8,418	3,227	2,428	1,877	11,969	2,836	2,099
	rm-8.4	44,459	8,507	8,467	8,461	12,087	5,008	5,109	33,171	5,097	5,057
	sort-8.16	88,068	14,893	14,843	15,834	60,631	61,739	21,948	119,150	18,711	7,914
	tar-1.14	163,296	20,411	20,713	20,592	115,234	95,765	77,910	200,394	14,384	12,095
	uniq-8.16	63,861	14,350	14,262	14,354	21,672	23,177	19,124	25,886	4,228	3,669
	Mean	65,151	15,080	15,152	15,235	43,827	28,505	21,782	72,140	11,803	10,686

Table 4.5: The final size, execution time and query number of ddmin, ProbDD and CDD on BM_{XML}. The last row shows the overall average across all three benchmark suites.

	Benchmark	Original size (#)	Final size (#)			Execution time (s)			Query number		
			ddmin	ProbDD	CDD	ddmin	ProbDD	CDD	ddmin	ProbDD	CDD
BM _{XML}	xml-071d221-1	20,090	10	15	20	73	114	144	29	50	69
	xml-071d221-2	20,387	13	14	20	155	146	243	60	69	117
	xml-1e9bc83-1	20,327	38	49	24	491	522	284	235	236	115
	xml-1e9bc83-2	20,222	79	78	76	1,391	814	867	725	390	384
	xml-1e9bc83-3	20,219	69	70	72	1,313	893	889	619	416	404
	xml-1e9bc83-4	19,985	156	139	143	3,911	1,939	2,521	1,943	935	1,173
	xml-1e9bc83-5	20,579	81	73	75	1,355	929	882	746	485	428
	xml-1e9bc83-6	19,880	127	126	124	3,563	1,852	1,548	1,907	964	749
	xml-1e9bc83-7	20,297	111	114	111	3,419	1,684	2,330	1,757	827	931
	xml-1e9bc83-8	20,327	100	107	100	2,862	1,636	1,446	1,451	751	592
	xml-1e9bc83-9	20,330	128	73	128	2,850	1,230	2,494	1,437	561	832
	xml-2d4ec80-1	20,129	76	72	76	570	522	791	384	304	354
	xml-327c8af-1	20,207	55	55	55	864	625	958	527	319	392
	xml-3398ac2-1	20,414	45	44	48	345	383	605	225	192	268
	xml-3398ac2-2	19,290	48	48	48	613	492	578	358	255	238
	xml-3398ac2-3	20,222	62	62	62	632	574	764	347	276	306
	xml-3398ac2-4	19,913	111	112	111	1,419	1,197	1,722	802	584	680
	xml-3398ac2-5	20,477	44	38	44	850	531	633	507	269	261
	xml-4c99b96-1	20,513	80	78	80	1,427	1,126	1,385	699	452	438
	xml-4c99b96-10	20,522	46	47	46	1,012	760	873	443	299	256
	xml-4c99b96-11	19,901	53	54	53	868	661	791	357	252	236
	xml-4c99b96-12	19,775	102	104	102	2,429	1,592	2,422	1,077	626	773
	xml-4c99b96-13	20,114	60	61	60	1,132	870	989	494	335	309
	xml-4c99b96-14	19,970	102	103	102	2,147	1,687	1,534	987	600	504
	xml-4c99b96-15	20,138	46	46	46	843	788	729	381	297	244
	xml-4c99b96-16	20,126	67	68	70	1,366	1,189	1,125	658	414	409
	xml-4c99b96-17	20,210	67	68	70	1,360	1,310	1,120	658	414	409
	xml-4c99b96-18	20,390	28	28	25	724	492	410	324	164	137
	xml-4c99b96-19	20,192	81	82	81	1,729	1,798	1,895	793	540	710
	xml-4c99b96-2	20,750	53	54	53	878	1,017	912	362	281	318
	xml-4c99b96-3	20,015	67	68	67	1,102	1,254	900	483	357	337
	xml-4c99b96-4	20,201	67	67	67	1,097	1,238	860	482	365	328
	xml-4c99b96-5	20,279	63	62	60	1,170	1,155	1,029	515	347	364
	xml-4c99b96-6	19,973	81	82	87	1,258	1,338	1,056	511	381	441
	xml-4c99b96-7	19,973	115	115	115	3,399	2,560	1,535	1,485	811	676
	xml-4c99b96-8	20,579	42	41	42	926	810	687	422	254	323
	xml-4c99b96-9	20,075	53	53	53	884	940	650	368	278	265
	xml-8ede045-1	20,192	48	61	69	1,152	1,232	1,137	499	389	487
	xml-8ede045-2	20,177	17	22	34	316	335	360	132	107	164
	xml-8ede045-3	20,393	31	31	31	476	431	313	199	134	134
	xml-8ede045-4	20,123	31	25	31	578	472	435	250	151	181
	xml-8ede045-5	20,051	17	20	17	185	246	204	61	72	80
	xml-8ede045-6	20,636	73	80	76	1,365	1,479	1,545	552	469	700
	xml-8ede045-7	20,054	106	106	106	2,851	1,686	1,237	1,447	614	634
	xml-8ede045-8	20,177	76	78	76	1,397	1,327	1,060	597	449	475
	xml-f053486-1	20,030	10	10	10	101	134	76	31	41	28
	Mean	20,190	56	56	58	972	819	821	453	314	327
All	Mean	31,989	233	235	237	2,999	2,189	2,102	2,752	1,309	1,320

probability p_0 , specifically, 0.1 for BM_C and BM_{DBT} , and $2.5\text{e-}3$ for BM_{XML} . The detailed results are shown in Table 4.4 and Table 4.5.

Efficiency and Effectiveness. Through our reproduction study, we find that the performance of ProbDD aligns with the results reported in the original paper, showing that ProbDD is significantly more efficient than ddmin. Across three benchmark suites, ProbDD requires 27.01% less time and 52.44% fewer queries, with p-value being $3\text{e-}09$ and $9\text{e-}14$, respectively. Moreover, we assess the effectiveness by measuring the sizes of the final minimized results. The effectiveness of ddmin and ProbDD varies across benchmarks. However, the Wilcoxon signed-rank test yields a p-value of 0.32, which is substantially higher than the 0.05 significance level. Therefore, we fail to reject the null hypothesis, suggesting no statistically significant difference in overall performance between the two algorithms.

4.4.5 Impact of Randomness in ProbDD

Finding 3: Randomness has no significant impact on the performance of ProbDD.

In ProbDD, elements with different probabilities are sorted accordingly, while elements with the same probability are randomly shuffled. However, randomness alone intuitively does not ensure a higher probability of escaping local optima and the effect of this randomness on performance has not been thoroughly investigated.

To this end, we conduct an ablation study by removing such randomness, creating a variant called ProbDD-no-random. We evaluate this variant across all benchmarks. The results indicate that the randomness does not significantly impact performance. Specifically, in terms of final size, execution time, and query number, ProbDD-no-random achieves 236, 2,069, and 1,238 compared to 235, 2,189, and 1,309 of ProbDD, respectively. The p-values of 0.87, 0.15, and 0.10 all exceed 0.05, so we fail to reject the null hypothesis, indicating no statistically significant differences.

4.4.6 Bottleneck Overcome by ProbDD

Finding 4: On tree-structured inputs, inefficient deletion attempts on complements and repeated attempts account for the bottlenecks of ddmin, which are overcome by ProbDD.

In the study of ProbDD, the authors demonstrate that ProbDD is more efficient than the baseline approach (ddmin) in tree-based reduction scenarios, where the inputs are

parsed into tree representations before reduction. Therefore, to uncover the root cause of this superiority, we follow the same application scenario and analyze the behavior of ProbDD in reducing the tree-structured inputs.

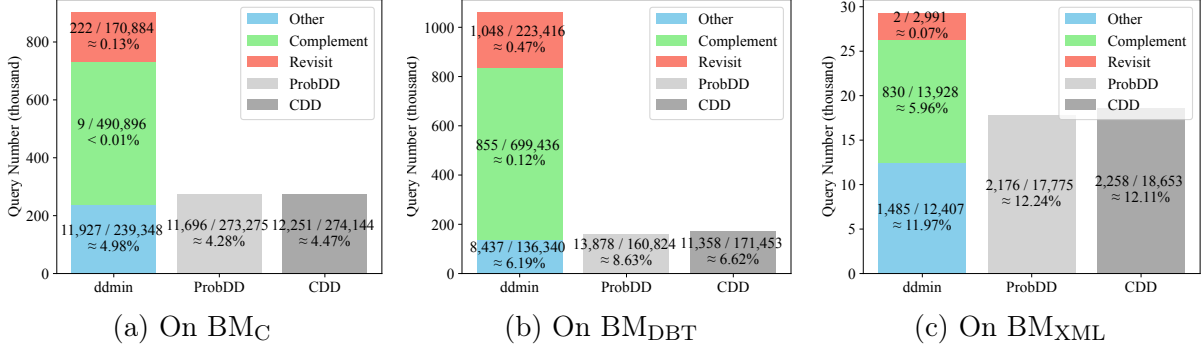


Figure 4.2: Visualization of queries within ddmin, ProbDD and CDD. In ddmin, three types of queries are displayed via stacked bars, the height of which denotes the query number. Within each bar, the number of successful queries, total queries and the corresponding success rate are annotated.

To further understand why ProbDD is more efficient than ddmin, we conduct in-depth statistical analysis on the query number (number of deletion attempts). Intuitively, performance bottlenecks lie in those queries with low success rates, impairing ddmin’s efficiency. Existing studies [49, 39] also demonstrate the presence of queries with low success rates. Therefore, to qualitatively and quantitatively identify the exact bottlenecks impairing ddmin, we statistically analyze all the queries in ddmin and categorize them into three types:

1. *Complement*: Queries attempting to remove the complement of a subset. According to ddmin algorithm, given a subset (smaller than half of the list L), it attempts to remove either the subset or its complement. However, evidence [49] shows that keeping a small subset and removing its complement is not likely to succeed, especially on structured inputs like programs.
2. *Revisit*: Queries attempting to remove the previously tried subset. After removing a subset, ddmin restarts the process from the first subset, leading to repeated deletion attempts on earlier subsets. Although the removal of one subset may allow another subset to be removable, such repetitions rarely succeed and thus offer limited improvement for the reduction [39].
3. *Other*: All other queries.

In addition to categorizing queries in ddmin into the above types, we also calculate the success rate of each type, aiming to reveal the bottlenecks of ddmin. Fig. 4.2 illustrates the distribution of queries for all types within ddmin, as well as the query number for ProbDD across all three benchmark suites.

On all benchmark suites, the number of successful queries in ddmin and ProbDD is remarkably similar, especially when contrasted with the substantial difference in the total number of queries. Specifically, on BM_C , ddmin achieves $222+9+11,927 = 12,158$ successful queries, closely matching the 11,696 successful queries from ProbDD. Similarly, on BM_{DBT} and BM_{XML} , ddmin performs $1,048 + 855 + 8,437 = 10,340$ and $2 + 830 + 1,485 = 2,317$ successful queries, respectively, both closely aligning with the 13,878 and 2,176 successful queries achieved by ProbDD. Besides, ddmin always performs significantly more failed queries, resulting in a larger total query number and thus a longer execution time, as previously discussed in § 4.4.4.

On all benchmark suites, a large portion of ddmin’s queries is categorized as *Complement* and *Revisit*; however, they both have a notably low success rate. For instance, on BM_C , out of a total of 901,128 queries, *Complement* and *Revisit* account for 490,896 (54.48%) and 170,884 (18.96%), respectively. Within such queries in *Complement* and *Revisit*, merely 9 (<0.01%) and 222 (0.13%) queries succeed, *i.e.*, only a tiny portion of attempts successfully reduce elements. These success rates are far less than those of queries within *Other* (4.98%), as well as those of ProbDD (4.28%). On the other benchmark suites, a similar phenomenon is observed.

Queries within *Complement* and *Revisit* categories constitute a large portion yet prove to be largely inefficient, wasting a significant amount of time and resources. On the contrary, those in *Other* achieve a much higher success rate, on par with that of ProbDD, and are responsible for most of the successful deletions. Therefore, we believe that these two categories, where queries are inefficient, are the main bottlenecks behind ddmin’s low efficiency. However, these bottlenecks are absent in ProbDD, as it does not consider complements of subsets and previously tried subsets for deletion.

4.4.7 1-Minimality of ProbDD?

Finding 5: Improving efficiency by avoiding ineffective attempts presents a trade-off by not ensuring 1-minimality, while such limitation can be mitigated by iteratively running the reduction algorithm until a fixpoint is reached.

Although ProbDD avoids *Revisit* queries to enhance efficiency, some reduction potentials

may be missed, as the deletion of a certain subset may enable a previously tried subset to become removable. Therefore, a limitation of ProbDD lies in that it increases efficiency by sacrificing 1-minimality. To substantiate this limitation, we examine how frequently ProbDD generates a list that is not 1-minimal, *i.e.*, can be further reduced by removing a single element. For instance, statistical analysis on BM_C reveals that among 6,871 invocations of ProbDD, 76 of them fail to generate a 1-minimal result, accounting for 1.1%. For these failed invocations, an average of 1.49 elements (tree nodes) can be further removed via single-element deletion.

However, such limitation is not apparent across all benchmark suites, as the results from ProbDD are not consistently larger than those from `ddmin`. Our further investigation reveals that these benchmarks are reduced on wrapper frameworks `Picireny` and `Chisel`. Both frameworks employ iterative loops to achieve a fixpoint, effectively reducing some elements missed in the first iteration.

4.5 Implications: A Counter-Based Model

Building on the aforementioned demystification of ProbDD, we discover that probability can be optimized away, and subset size can be pre-computed. Hence, we propose **Counter-Based Delta Debugging** (CDD), to reduce the complexity of both the theory and implementation of ProbDD, and validate the correctness of our prior theoretical proofs.

Subset Size Pre-calculation. Based on lemma 4.3.3 in § 4.3.2, the size for each round can be pre-calculated. Therefore, as shown at line 12 – line 15 in Algorithm 6, we utilize the current round r and the initial probability p_0 to determine the subset size s . The size of the selected subset decreases as the round counter increases. This is intuitively reasonable since, after a sufficient number of attempts on a large size have been made, it becomes more advantageous to gradually reduce the subset size for future trials. Furthermore, this trend aligns well with that of ProbDD, in which probabilities of elements gradually increase, resulting in a smaller subset size.

Main Workflow. The simplified ProbDD is illustrated in Algorithm 6, from line 1 to line 11. Before each round, the CDD pre-calculates the subset size on line 3 and then partitions L using this size on line 4. Then, similar to `ddmin`, it attempts to remove each subset on line 5 – line 8. The subset size continuously decreases until it reaches 1, meaning that each element will be individually removed once.

Revisiting the Running Example. Returning to Table 4.3, under the same conditions, CDD achieves the same results as ProbDD but without the need for probability calculations.

Algorithm 6: CDD (L, ψ)

Input: L : a list of element to be reduced.

Input: $\psi : \mathbb{P} \rightarrow \mathbb{B}$: the property to be preserved by L .

Input: p_0 : the initial probability given by the user.

Output: the minimized list that still exhibits the property ψ .

```
1  $r \leftarrow 0$  // The round number, initially 0.
2 do
    // Compute subset size by round number
3    $s \leftarrow \text{ComputeSize}(r, p_0)$ 
    /* Partition L into subsets with  $s$  elements. If it does not divide
       evenly, leave a smaller remainder as the final subset. */
4   subsets  $\leftarrow$  Partition( $L, s$ )
5   foreach subset  $\in$  subsets do
6     temp  $\leftarrow L \setminus$  subset
       // Remove subset if it is removable
7     if  $\psi(\text{temp})$  is true then
8        $L \leftarrow \text{temp}$ 
       // Update the  $r$  and move to next round.
9    $r \leftarrow r + 1$ 
10 while  $s > 1$ 
11 return  $L$ 
12 Function ComputeSize( $r, p_0$ ):
    Input:  $r$ : the current round number.
    Input:  $p_0$ : the initial probability given by the user.
    Output: The size of the subset to be used in the current round.
    // Calculate the estimated probability of round  $r$ 
13    $p_r \leftarrow p_0 \times 1.582^r$ 
    // Calculate corresponding subset size of round  $r$ 
14    $s_r = \arg \max_{s \in \mathbb{N}^+} s \times (1 - p_r)^s$ 
15   return  $s_r$ 
```

This is because both the probability and subset size s can be directly determined from the round number r .

Evaluation. As shown in Table 4.4 and Table 4.5, CDD outperforms ddmin *w.r.t.* efficiency, with 29.91% less time and 52.04% fewer queries. Meanwhile, CDD performs comparably to ProbDD *w.r.t.* final size, execution time and query number, with a p-value of 0.42, 0.29 and 0.70, respectively, indicating insignificance between these two algorithms. CDD is expected to perform on par with ProbDD since it is designed to provide further insight and simplify the intricate design of ProbDD, rather than to surpass its capabilities. Furthermore, its comparable performance to ProbDD further validates the non-necessity of randomness and our assumption in lemma 4.3.1.

Bottleneck and 1-minimality. Revisiting the bottlenecks presented in Fig. 4.2, CDD possesses a query number and success rate close to those of ProbDD, indicating that CDD also overcomes the bottlenecks of ddmin. Additionally, similar to ProbDD, 1-minimality is absent in CDD, although iterations help mitigate this issue.

Finding 6: CDD always achieves comparable performance to ProbDD, which further supports our previous findings, including the theoretical simplifications regarding size and probability, analysis of randomness, bottlenecks, and 1-minimality.

4.6 Limitations and Threats to Validity

In this section, we discuss the limitations of CDD, and potential factors that may impair the validity of our experimental results.

4.6.1 Limitations

As discussed in § 4.4.7, compared to ddmin, neither ProbDD nor CDD guarantees 1-minimality. This limitation arises because after successfully removing a subset, ddmin restarts the process from the first subset, whereas ProbDD and CDD continue from the next subset, skipping all the previously tried subsets. Therefore, although ProbDD and CDD complete the reduction process more quickly, they may miss certain reduction opportunities and produce larger results than ddmin.

However, reduction and debloating tools generally invoke these reduction algorithms in iterative loops until a fix-point is reached, gradually refining the results and mitigating

limitations, as mentioned in § 4.4.7. Table 4.4 and Table 4.5 further support this point by showing that with multiple iterations, ProbDD and CDD achieve significantly higher efficiency compared to ddmin, while still producing results that are comparable to ddmin w.r.t. effectiveness.

4.6.2 Threats to Validity

For internal validity, the main threat comes from the potential impact from the assumption, as discussed in § 4.3.1. Specifically, without assuming that the number of elements in L is always divisible by the current subset size, we could not further refine the mathematical model of ProbDD to achieve a simpler representation. However, such an assumption might impact the actual performance, potentially negating the benefits of our simplification. To this end, we conduct extensive empirical experiments, demonstrating that CDD, the simplified algorithm derived from this assumption, exhibits no significant difference from ProbDD.

For external validity, the threat lies in the generalizability of our findings across application scenarios. To mitigate this threat, we perform experiments on 76 benchmarks, including C programs triggering real-world compiler bugs, XML inputs crashing XML processing tools, and benchmarks from software debloating tasks. These benchmarks have covered various use scenarios of minimization algorithms.

4.7 Related Work

In this section, we discuss related work of test input minimization around three aspects: effectiveness, efficiency, and the utilization of domain knowledge.

Effectiveness. Test input minimization is an NP-complete problem, in which achieving the global minimum is usually infeasible. Therefore, existing approaches to improving effectiveness mainly aim to escape local minima by performing more exhaustive searches. Since enumerating all possible subsets is infeasible, Vulcan [143] and C-Reduce [99] enumerate all combinations of elements within a small sliding window, and exhaustively attempt to delete each combination, resulting in smaller final program sizes. In contrast, ProbDD and CDD do not exhibit clear actions targeted at breaking through local optima, suggesting they cannot achieve better effectiveness than ddmin, as aligned with our evaluation in § 4.4.

Efficiency. If parallelism is not considered, the core of boosting efficiency is the enhanced capability to avoid relatively inefficient queries. For example, Hodovan and

Kiss [49] proposed disregarding attempts to remove the complement of subsets, the success rate of which is unacceptably low in some scenarios. Besides, Gharachorlu and Sumner [39] proposed One Pass Delta Debugging (OPDD), which continues with the subset next to the deleted one, rather than starting over from the first subset. This optimization also avoids some redundant queries in ddmin, reducing runtime by 65%. As revealed by our analysis, these two above-mentioned optimizations are implicitly incorporated within ProbDD and CDD, and thereby contributing to their higher efficiency than ddmin.

Utilization of Domain Knowledge. There is an inherent trade-off between effectiveness and efficiency in test input minimization. For the same algorithm, achieving a better result, *i.e.*, a smaller local optimum, requires more queries to be spent on trial and error. However, employing domain knowledge [99, 55, 56, 148] can still improve the overall performance. For instance, J-Reduce is both more effective and efficient than HDD in reducing Java programs, as it escapes more local optima by program transformations while simultaneously avoiding more inefficient queries via semantic constraints, leveraging the semantics of Java. Our analysis on ProbDD indicates that the probabilities primarily function as counters and do not utilize or effectively learn the domain knowledge of an input. Besides, the evaluation on CDD, a simplified algorithm without utilizing probability, demonstrates that prioritizing elements via such probabilities does not yield significant benefits, thus validating our analysis.

4.8 Conclusion

This chapter conducts the first in-depth analysis of ProbDD, which is the state-of-the-art variant of ddmin, to further comprehend and demystify its superior performance. With theoretical analysis of the probabilistic model in ProbDD, we reveal that probabilities essentially serve as monotonically increasing counters, and propose CDD for simplification. Evaluations on 76 benchmarks from test input minimization and software debloating confirm that CDD performs on par with ProbDD, substantiating our theoretical analysis. Furthermore, our examination on query success rate and randomness uncovers that ProbDD’s superiority stems from skipping inefficient queries. Finally, we discuss trade-offs in ddmin and ProbDD, providing insights for future research and applications of test input minimization algorithms.

Chapter 5

LPR: Large Language Models-Aided Program Reduction

This chapter investigates how large language models (LLMs) can improve the effectiveness and generality of program reduction. It utilizes the semantic transformation capabilities of LLMs to refine reduction results, thereby narrowing the gap between language-specific and language-general approaches.

5.1 Introduction

Program reduction has been widely used in various software engineering tasks [28, 46], especially in compiler testing and debugging [79, 127, 68, 116, 115], but a critical challenge in program reduction has not been properly addressed, *i.e.*, the trade-off between generality across languages and specificity to a certain language. Currently, there are two categories of program reduction techniques: language-specific reduction [99, 55, 56] and language-generic reduction [145, 86, 109, 143, 151]. The former category leverages language-specific semantics to transform and shrink programs in certain languages, while the latter only uses syntax to guide parse tree level pruning and transformations, which is applicable to any programming language.

Language-specific reducers are usually more effective in reduction, but designing an effective reducer for a specific language, especially language-specific transformations, requires a deep understanding of language features and a significant amount of time and engineering effort. Therefore, only a limited set of languages have custom-designed reducers, such as

C [99], Java [55, 56], and SMT-LIBv2 [92]. Meanwhile, language-generic reducers can be applied to diverse languages, but lack the knowledge of language features and semantics and thus are incapable of performing language-specific transformations (*e.g.*, function inlining) that can enable further reduction. As a result, they cannot utilize particular features of each language to produce optimally reduced programs.

This study strives to find a sweet spot between generality across languages and specificity to a certain language, by synergistically combining the strengths of both categories of program reduction techniques. Specifically, we notice that the major limitation of language-generic reducers lies in their incapability to perform language-specific transformations. Language-generic reducers such as Perses stand out as high generality when reducing programs across various programming languages, while they lack awareness of semantic information to achieve further progress. If we could help language-generic reducers conquer this limitation, they are likely to produce smaller results.

Meanwhile, we also notice that recent Large Language Models (LLMs) could be powerful assistants in performing language-specific transformations, like how they perform in other software engineering tasks such as code generation and test generation [110, 54, 130, 27, 35, 138, 44]. Specifically, LLMs are trained with massive programs, and they have started to exhibit the ability to analyze and transform programs in prevalent languages. If we can properly leverage this ability for program reduction, we may have a language-generic reducer being aware of the semantics of various prevalent languages. In addition, the utilization of LLMs can simplify the customization and extension of reducers, as it would be time-consuming and labor-intensive to manually implement a language-specific reducer or add functionality to an existing one (such as C-Reduce using the Clang frontend to implement C/C++-specific program transformations).

Challenges of Using LLMs. LLMs are not the silver bullet to program reduction. When LLMs are instructed to handle source code as inputs, due to the intrinsic limitations of LLMs [34, 125, 71], LLMs may be unable to understand subtle differences in code [75] and be distracted by irrelevant context [105]. Moreover, in program reduction, input programs typically contain tens of thousands of line of code, which surpasses the input limits of LLMs. Besides, without effective guidance, LLMs are unclear about what transformations to perform.

LLMs-Aided Program Reduction (LPR). We propose LPR in this chapter, which is, to the best of our knowledge, the first approach that integrates LLMs for program reduction tasks. LPR synergistically leverages the strengths of both language-generic program reducers and LLMs. Specifically, LPR alternates between invoking a language-generic reducer (we use Perses in experiments) and the LLM. Initially, Perses efficiently

reduces large programs to a size manageable for the LLM. Subsequently, the LLM further transforms Perses’s output based on specific user-defined prompts that dictate the required transformations. Following this, Perses is re-invoked, as transformations made by the LLM often create additional opportunities for reduction. This process iterates until the program cannot be further minimized. For transformations, we have identified five language-generic transformations to enable further reduction: *Function Inlining*, *Loop Unrolling*, *Data Type Elimination*, *Data Type Simplification*, and *Variable Elimination*.

To address the aforementioned challenge of using LLMs, LPR is designed with a multi-level prompting approach. In detail, LPR initially requests the LLM to identify a list of potential targets for a given transformation, and then sequentially instructs the LLM to apply the transformation on each target. The multi-level prompt guides the LLM in a more concentrated way, by excluding irrelevant context and other targets that may distract the LLM.

We have conducted extensive evaluations on LPR, illustrating its superiority over Vulcan, the state-of-the-art language-generic algorithm. On three benchmark suites, *i.e.*, Benchmark-C, Benchmark-Rust and Benchmark-JS, LPR produces smaller programs than Vulcan by 24.93%, 4.47% and 11.71%, separately. Moreover, LPR and Vulcan complement each other. For C language which C-Reduce is optimized for, by applying Vulcan to the output produced by LPR, we attain program sizes that are on par with those achieved by C-Reduce. For reduction efficiency, LPR performs comparably to Vulcan. In terms of the execution time perceived by users, LPR is more efficient than Vulcan on reducing complex programs. Furthermore, our detailed analysis indicates that each of the proposed transformations plays a crucial role in the reduction process. We also compare the LPR’s performance with the multi-level prompt against that without it, illustrating the efficacy of our proposed multi-level prompting approach.

Contribution. This study makes the following contributions.

- We introduce LPR, the first attempt to use LLMs for program reduction. By synergizing the capabilities of both language-generic program reducers and LLMs, LPR exhibits both generality across various languages and the awareness of semantics in specific languages. Moreover, LPR is easy and flexible to extend with new transformations by simply designing new prompts.
- We propose a multi-level prompting approach to guide LLMs to execute program transformations, and demonstrate its effectiveness in practice. We propose five general-purpose transformations for LLMs to reduce programs or create more reduction opportunities for language-generic program reducers to exploit.

- We comprehensively evaluated LPR on 50 benchmarks across three commonly used languages: C, Rust and JavaScript. Results demonstrate LPR’s strong effectiveness and generality.

5.2 Background

5.2.1 Program Reduction

Given a program P with a certain property, *e.g.*, triggering a compiler bug, the goal of program reduction is to search for a minimal program P_{min} that still triggers the bug. Program reduction has demonstrated its significant usefulness in removing bug-irrelevant code snippets. The original bug-triggering code [144, 64, 107, 79] may have thousands of lines, whereas the distilled version from program reduction tools only contains dozens of lines of code [108]. Some algorithms are designed to generalize across multiple programming languages, while others are customized for certain languages.

Language-Generic Reducers

Some reducers can be generalized to multiple languages. For instance, given the formal syntax of a programming language, algorithms like HDD and Perses can be used to reduce programs corresponding to this language. HDD parses the program into a parse tree and then applies the `ddmin` [145] at each level of the tree to remove unnecessary tree nodes as much as possible. Perses goes further than HDD by performing certain transformations on the formal syntax to avoid generating syntactically incorrect program variants. Vulcan extends Perses, by introducing novel auxiliary reducers to exhaustively search for smaller variants by replacing identifiers/sub-trees and deleting local combinations of tree nodes on the parse tree.

However, different languages possess their own unique semantic features. Although the aforementioned algorithms are relatively efficient [109], they are incapable of utilizing the unique semantics of a particular language to further reduce a program. For example, these algorithms lack the ability to perform transformations like function inlining. Although Vulcan can identify more reduction opportunities through transformations such as identifier replacement and local exhaustive search, its approach is akin to "brute force" enumeration. This method lacks awareness of the given program’s semantics, making it less effective and efficient overall.

Language-Specific Reducers

Previous work has introduced reducers customized for some specific languages. For example, C-Reduce [99] is the most effective reduction tool for C code. It comprises multiple passes that transform the program based on features of the language, thereby making it smaller. Language-specific reducers often rely on static program analysis tools for analysis and modification, *e.g.*, LibTooling [81] is employed in C-Reduce.

However, developing a language-specific reducer is nontrivial. To the best of our knowledge, only a few languages have specific reducers, such as C [99], Java [55, 56], and SMT-LIBv2 [92]. The reason is that the process of designing a reducer for a language, or adding new transformations to an existing reducer, is time-consuming and labor-intensive. For instance, in version 2.10.0 of C-Reduce [100], function inlining was implemented with 604 lines of C++ code. Such challenges impede the development and maintenance of language-specific reducers.

5.2.2 Large Language Models

Large Language Models refer to a type of deep learning models trained on huge data sets for diverse tasks. The advent of LLMs has opened up numerous potential opportunities across diverse research fields. LLMs are not only proficient in processing natural languages but also exhibit substantial capabilities in understanding and processing programming languages. This highlights the promising future and evolutionary prospects in the realm of software engineering. Recently, LLMs have been applied and assessed on various software engineering tasks, such as automatic program repair [53, 139, 138, 137, 35] and program generation [77, 156, 113].

However, despite the usefulness of LLMs, some researchers [75] illustrate that current LLMs are weak in distinguishing nuances between programs. Moreover, the memorizing and processing capacity of LLMs deteriorates as the input size grows [34, 103]. Besides, one cannot expect LLMs to automatically complete complex tasks; they must be guided accordingly [138, 75]. Therefore, for program reduction, directly asking LLMs to reduce programs with tens of thousands of lines is impractical.

5.3 Approach

In this section, we begin by introducing a motivating example that illustrates the core concepts and challenges addressed by our approach. Next, we provide a comprehensive

overview of the LPR workflow. We also outline the details of prompts and proposed transformations in the workflow that enable the LLM to function effectively on given programs.

5.3.1 Motivation

<pre> 1 2 // nested loop 3 for (i = 0; i < 7; i++) 4 for (j = 0; j < 5; j++) 5 for (k = 0; k < 7; k++) 6 fn8(ad[i][j][k], "g_643[i 7][j][k]", aj)); </pre> (a) Original	<pre> 1 2 // nested loop 3 for (i = 0; i < 7; i++) 4 for (j = 0; j < 5; j++) 5 for (k = 0; k < 7; k++) 6 s = s ^ ad[i][j][k]; 7 </pre> (b) LPR: Before the <i>Loop Unrolling</i> transformation	<pre> 1 2 // the nested loop is fully unrolled 3 // into hundreds of lines 4 s = s ^ ad[2][0][5]; 5 s = s ^ ad[2][0][6]; 6 s = s ^ ad[2][1][0]; 7 s = s ^ ad[2][1][1]; 8 </pre> (c) LPR: After the <i>Loop Unrolling</i> transformation
<pre> 1 2 // all lines except for the bug -triggering one 3 // is removed by Perses 4 s = ad[2][1][0]; 5 </pre> (d) Final result of LPR	<pre> 1 2 for (i = 0; i < 7; i++) 3 for (j = 0; j < 5; j++) 4 for (k = 0; k < 7; k++) 5 fn8(ad[i][j][k], "g_643[i 6][j][k]", aj); </pre> (e) Final result of Vulcan	<pre> 1 2 for (; h < 7; h++) { 3 j = 0; 4 for (; j < 5; j++) 5 printf("% 6 } 7 </pre> (f) Final result of C-Reduce

Figure 5.1: Code snippet from LLVM-31259, showcasing the original code, the effectiveness of *Loop Unrolling*, and the final results by LPR, Vulcan and C-Reduce.

A motivating example is displayed in Fig. 5.1, where the original code contains highly nested loops, shown in Fig. 5.1(a). From Fig. 5.1(b) to Fig. 5.1(c), the nested loops are fully unrolled into hundreds of lines via *Loop Unrolling*, based on the semantic transformations from the LLM. Despite the temporary size increase, the following Perses effectively eliminates all lines except for the bug-relevant one. This is also the final result of LPR, presented in Fig. 5.1(d). By contrast, as shown in Fig. 5.1(e), Vulcan is incapable of escaping the local minima by exhaustively replacing identifiers and tree nodes. Similarly, C-Reduce cannot fully break down the loop structures, given that it is not integrated with transformations to unroll loops. Even though loop unrolling techniques can be added into C-Reduce in future versions, it will be labor-intensive to implement a specific transformation compared to user-defined prompts in natural language.

5.3.2 Workflow

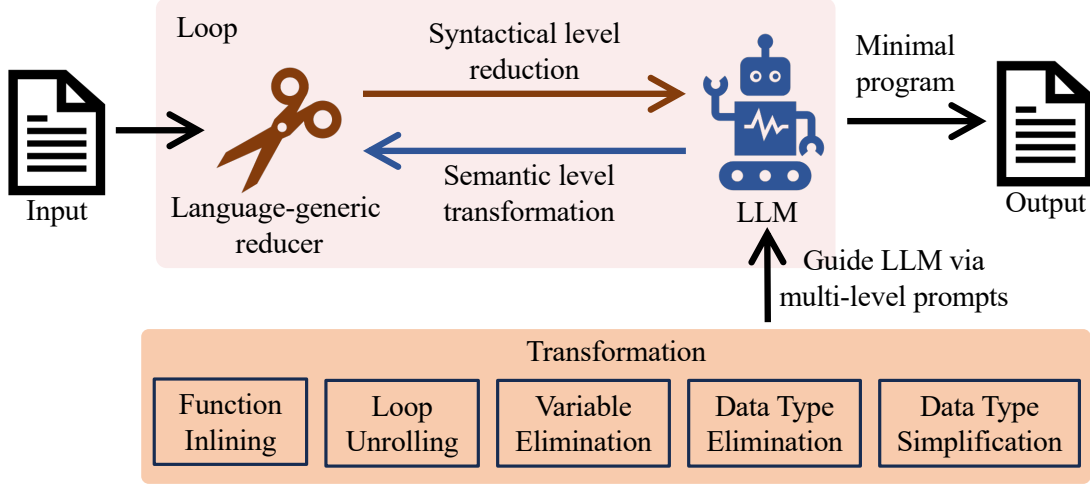


Figure 5.2: The workflow of LPR.

The overview of the workflow is outlined in Fig. 5.2. Given a bug-triggering program as input, LPR invokes a language-generic reducer and the LLM alternately, until the target program cannot be further reduced. In each iteration, the language-generic reducer efficiently reduces the given program to a minimal one. By contrast, the LLM leverages the semantic knowledge of the language and transforms the program given by the language-generic reducer. This process is guided by user-defined prompts, aiming to expose more reduction potentials to the language-generic reducer.

Algorithm 7 shows LPR’s reduction algorithm. Given as inputs (1) a program P targeted for reduction, (2) a property ψ that must be preserved, and (3) the pre-defined *prompts*, LPR generates a reduced program P_{min} . The pre-defined *prompts* consist of *primaryQuestion* and *followupQuestion*, in which *primaryQuestion* instructs the LLM to identify a list of targets to be transformed, and *followupQuestion* guides the LLM to apply transformation on each target individually. They will be further introduced in § 5.3.4. Initially, LPR calls Perses to quickly minimize P , so that the size of the program becomes manageable for the LLM to process. Next, LPR loads a sequence of transformations as delineated in line 4, and then iterates through each transformation, as detailed from line 5 to line 13. § 5.3.3 displays the details of each transformation.

During this process, for each transformation, the algorithm retrieves a predefined primary question along with a follow-up question on line 6 – line 7. LPR firstly asks the LLM the

Algorithm 7: LPR ($P, \psi, prompts$)

Input: P : the program to be reduced.
Input: $\psi : \mathbb{P} \rightarrow \mathbb{B}$: the property to be preserved by P .
Output: P_{min} : the reduced program that preserves the property.
// Quickly minimize P for LLM to process.

```
1  $P \leftarrow \text{Perses}(P, \psi)$ 
  // Monotonically minimize the size of  $P$  in the loop.
2 repeat
3    $P_{min} \leftarrow P$ 
4    $transformList \leftarrow \text{getTransformList}(prompts)$ 
  // Iterate through each transformation.
5   foreach  $transform \in transformList$  do
6      $primaryQuestion \leftarrow \text{getPrimaryQuestion}(transform)$ 
7      $followupQuestion \leftarrow \text{getFollowupQuestion}(transform)$ 
  // Ask LLM to identify a list of targets.
8      $targetList \leftarrow \text{getTargetList}(P, primaryQuestion)$ 
9     foreach  $target \in targetList$  do
10      // Let LLM apply the transformation on the target.
11       $P_{tmp} \leftarrow \text{applyTransformation}(P, followupQuestion, target)$ 
12      if  $\psi(P_{tmp})$  then
13        //  $P_{tmp}$  preserves the property.
14         $P \leftarrow P_{tmp}$ 
15      // Run Perses for further reduction.
16       $P \leftarrow \text{Perses}(P, \psi)$ 
17 until  $|P| \geq |P_{min}|$ 
18 return  $P_{min}$ 
```

primary question under the current program. This query aims to guide the LLM to generate a list of specific targets upon which the transformation will be executed. For instance, for *Loop Unrolling* in the motivating example Fig. 5.1(b), the LLM is asked to identify a list of loops in the given program to be unrolled in the *primaryQuestion*, and returns a target list `[for (i = 0; ...), for ( j = 0; ...), for ( k = 0; ...)]`.

On line 9 to line 12, LPR uses *followupQuestion* to guide the LLM to apply the transformation on each identified target within the program. In the motivating example, the *followupQuestion* can be framed as “Given the program { PROGRAM } and the loop `for(i=0;...)`, optimize it via loop unrolling”. The modified program is then extracted from the LLM’s response text on line 10. In the example, all loops are unrolled into repeated lines of code in Fig. 5.1(c).

In experimental scenarios, given an input program and the prompt, the LLM may generate multiple transformed programs, as the number of responses can be customized. Among all transformed programs, LPR keeps the smallest one that still passes the property test, and discards others. If no transformed program returned from this query satisfies the property, LPR keeps the original one before this query. After each transformation is completed, a language-generic reducer such as Perses [109] is employed to seek additional reduction opportunities, considering that the transformation might have introduced new potential for further simplification. The algorithm persists in the outermost loop until it reaches a fixpoint, signifying that the program size can no longer be reduced.

5.3.3 Transformations

To further search for reduction opportunities on a bug-triggering program via the LLM, we propose five general transformations to guide the LLM, *i.e.*, *Function Inlining*, *Loop Unrolling*, *Data Type Elimination*, *Data Type Simplification* and *Variable Elimination*.

Function Inlining. This transformation identifies a function and inlines it to eliminate all call sites of this function, and instead substitutes them with the corresponding function body. As functions are commonly found in bug-triggering programs, there is significant room for function inlining to reduce tokens or provide further reduction opportunities.

Loop Unrolling. Loop unrolling, also known as loop unwinding, is a widely used loop transformation approach to optimize the execution. In this task, it can also be employed to find more reduction opportunities. *Loop Unrolling* identifies a loop structure and attempts to unroll the loop into a code snippet repeating a single iteration. This is motivated by the fact that language-generic reducers may be incapable of dismantling or directly removing

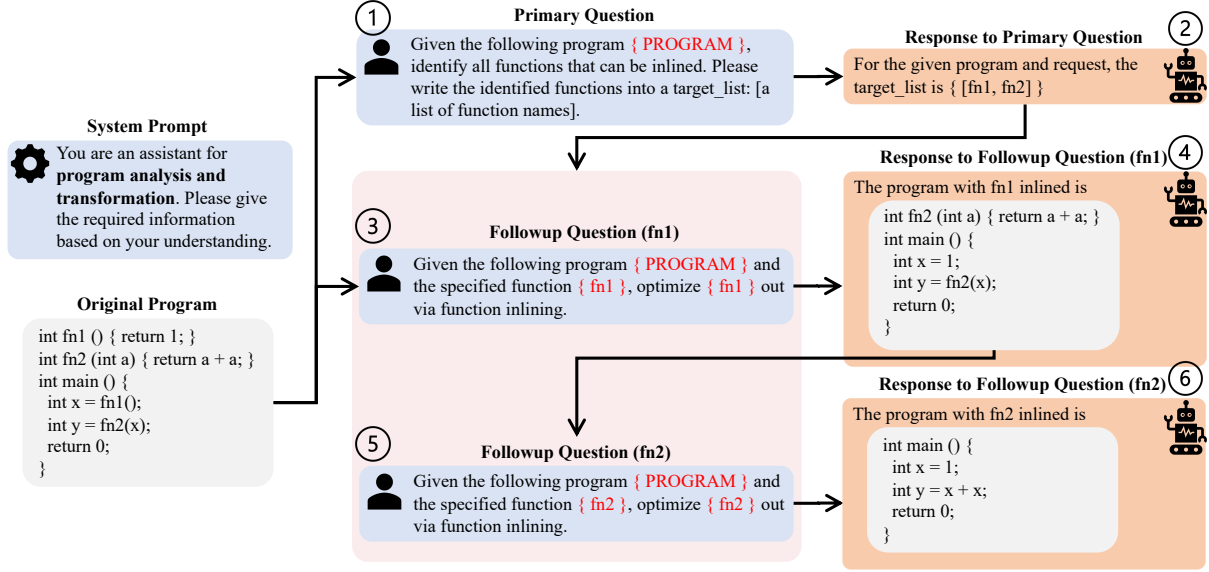


Figure 5.3: An example of prompt design. ⚙️ and 👤 denote system prompt and user prompt provided by the users. 🤖 denotes the responses from the LLM.

the loop structure, while they may be able to reduce the repeated code after loop unrolling, as shown in Fig. 5.1.

Data Type Elimination. Some data types in bug-triggering programs may be irrelevant to the bug, such as identifiers defined by `typedef` in C, and type alias created by `type` keyword in Rust. We propose *Data Type Elimination* to eliminate the alias and replace the occurrence of each alias with its associated original data type.

Data Type Simplification. In programs with complex data types, such as structures, arrays, and pointers, not all components are essential for maintaining bug-triggering properties. For example, a bug-triggering program containing a `struct` with three integer members can be simplified into three distinct integer variables, and possibly only one variable is essential. To facilitate this simplification, we introduce *Data Type Simplification*, a strategy designed to transform variables of complex data types into variables of primitive data types, like integers or floats.

Variable Elimination. Intermediate variables are pervasive in programs, and reducing them is desirable in program reduction tasks. Besides, some variables, although not being used, are hard to eliminate. For instance, to remove an unused parameter, both the parameter defined in the function and its corresponding argument passed to the call site

of this function should be removed simultaneously. This is hard or even impossible for language-generic reduction tools. Therefore, we propose *Variable Elimination* to optimize out both intermediate and unused variables.

The proposed transformations are universally applicable across various programming languages, offering a broad utility. By performing these transformations on programs via the LLM, substantial human effort is saved from designing and implementing reducers that target these transformations. While certain existing language-specific reducers like C-Reduce, may already incorporate some of these transformations, *e.g.*, *Function Inlining* and *Variable Elimination*, creating new transformation passes remains a non-trivial task for users. Our approach not only simplifies this process but also extends its reach across multiple programming languages.

5.3.4 Multi-Level Prompts

Prompts enable LLMs to apply the transformations mentioned above. We take *Function Inlining* as an example. We avoid directly instructing the LLM to perform transformations exhaustively, such as inlining all functions in a program in a single query, which might overwhelm its processing capabilities, especially for programs with multiple functions. Instead, we employ a multi-level prompting approach. Fig. 5.3 presents an example of *Function Inlining*. First, we pose a primary question to the LLM (step ①): “Given the following program { PROGRAM }, identify all functions that can be inlined.” Based on the list provided by the LLM (step ②), we then ask a series of follow-up questions (step ③ and step ⑤) like “Given the following program { PROGRAM } and the specified function { fn1 }, optimize { fn1 } out via function inlining.”, and the LLM do the transformations accordingly (step ④ and step ⑥). This strategy excludes irrelevant context and ensures that the queries are more targeted, thereby increasing the likelihood of the LLM generating high-quality results. For other transformations, the prompts follow a similar template — first prompting the LLM to identify a target list, and then instructing it to attempt optimization of each target.

5.4 Evaluation

In this section, we evaluate the reduction effectiveness and efficiency of LPR. Specifically, we conducted the following research questions.

RQ1 What is the effectiveness of LPR in program reduction?

RQ2 To what extent is the efficiency of LPR in program reduction perceived by users?

RQ3 What is the effectiveness of each transformation in LPR?

5.4.1 Experimental Setup

Within the workflow of LPR, we employ Perses [109] as the language-agnostic reducer due to its superior efficiency compared to Vulcan. Additionally, we utilize OpenAI API [1], specifically the gpt-3.5-turbo-0613 version, to serve as the LLM. We also develop a variant named LPR+Vulcan, which invokes Vulcan to further reduce the program after LPR finishes. The experiments are conducted on an Ubuntu 22 server with an Intel(R) Xeon(R) Gold 6348 CPU @ 2.60GHz and 512 GB RAM. For a fair comparison, all algorithms were executed in a single-process, single-thread environment.

Benchmarks. To measure the effectiveness and efficiency of LPR across various languages, we employ three benchmark suites: Benchmark-C, Benchmark-Rust and Benchmark-JS. The Benchmark-C, previously collected and utilized by previous studies [143, 118, 151], comprises 20 large complex programs triggering real-world bugs in LLVM or GCC. Benchmark-Rust, incorporating 20 bug-triggering Rust programs, has also been used in prior research [143]. We further craft Benchmark-JS, a non-public benchmark suite, for this study. Specifically, we use FuzzJIT [126] to fuzz a prevalent JavaScript engine, *i.e.*, JavaScriptCore (version c6a5bcc), and then randomly collect 10 programs that cause miscompilations in JIT compiler. Since the programs and reduced programs in Benchmark-JS are not publicly accessible at the time of LLMs’ training, and thus not in the training sets of LLMs. The evaluation on Benchmark-JS helps to investigate whether LPR suffers from the data leakage problem [135]. In total, the evaluation benchmarks encompass 50 programs triggering real-world bugs in compilers, spanning across three popular programming languages.

Baselines. In all three benchmark suites, we use Perses and Vulcan as baselines. Perses stands out as a highly effective and efficient program reduction tool. To avoid the occurrence of syntactical invalid variants during the reduction process, it transforms and normalizes the formal syntax of a programming language. Vulcan [143], building upon Perses, provides three manually designed auxiliary reducers to further search for reduction opportunities on results from Perses. Compared to Perses, Vulcan achieves a reduction in the number of tokens, albeit at the expense of increased running time. They are both language-generic and are applicable across a broad spectrum of programming languages. We also include C-Reduce (v2.9.0) as an additional baseline. C-Reduce not only stands as the most effective

algorithm for C, it can also be applied to other languages, though not customized for them.

Configuration. If not otherwise specified, our experiments are conducted by invoking OpenAI API (version gpt-3.5-turbo-0613), with the proposed multi-level prompts and transformations in § 5.3.3 and § 5.3.4. To effectively harness the inherent randomness of LLMs, we set `temperature=1.0`. This high value encourages the LLM to generate more diverse outcomes [3]. Additionally, we employ `n=5` to generate five distinct results for every query [2], enabling us to choose the smallest program preserving the property ψ as the optimal result. All the rest configurations are set to their default values.

5.4.2 RQ1: Reduction Effectiveness

We measure the effectiveness of LPR, LPR+Vulcan and baseline algorithms via the final program sizes in tokens. A smaller size is favored, as it signifies the removal of more bug-irrelevant code, thereby saving developers more manual effort. The effectiveness of each algorithm on all three benchmark suites is presented in Table 5.1. Due to the randomness of LPR, we repeat five times for LPR and LPR+Vulcan on every benchmark, and display the mean value and standard deviation value in the table. On each benchmark, the minimal results are highlighted in bold.

Benchmark-C. On this benchmark suite, Perses reduces the programs to an average of 247.8 tokens. Building upon this, Vulcan further compresses the average program size into 157.4, *i.e.*, thereby continues to decrease the program size by 34.35%. Despite Vulcan’s notable reduction progress, LPR is still capable of continuing to push the limit of Perses, and reduces the programs in Benchmark-C into 105.8 tokens on average across all five runs. It cuts down the average program size of Perses by 51.33%, outperforming Vulcan significantly by 24.93% (proved by a p-value of 0.002). C-Reduce stands out by achieving the lowest average program size, *i.e.*, 85.7 tokens. This performance is anticipated as C-Reduce incorporates various transformation passes specifically designed for C. Despite relying on only general transformations, LPR+Vulcan still achieves performance comparable to C-Reduce, averaging 86.1 tokens. Moreover, it outperforms C-Reduce in 13 out of 20 benchmarks, highlighting its effectiveness with only language-generic transformations.

Benchmark-Rust. On Benchmark-Rust, Perses and Vulcan produce programs with 212.5 and 184.2 tokens on average, separately. LPR and LPR+Vulcan further shrink the average program size into 147.5 and 135.5 tokens. C-Reduce produces the second-largest programs on average, only smaller than Perses. This is anticipated since C-Reduce lacks specialized transformations for Rust.

Table 5.1: The reduction sizes of Perses, Vulcan, C-Reduce, LPR and LPR+Vulcan. Best results among all algorithms are highlighted in bold font.

	Benchmark	Original	Perses	Vulcan	C-Reduce	LPR	LPR+Vulcan
Benchmark-C	LLVM-22382	21,068	144	108	70	73.2 $\pm$ 1.6	69.8 $\pm$ 1.8
	LLVM-22704	184,444	78	62	42	43.6 $\pm$ 3.1	41.8 $\pm$ 3.3
	LLVM-23309	38,647	464	303	118	105.8 $\pm$ 9.3	91.2 $\pm$ 8.2
	LLVM-23353	30,196	98	91	74	68.8 $\pm$ 8.0	66.6 $\pm$ 6.5
	LLVM-25900	78,960	239	104	90	93.4 $\pm$ 11.1	84 $\pm$ 6.5
	LLVM-26760	209,577	120	56	43	62.8 $\pm$ 18.6	52.6 $\pm$ 5.4
	LLVM-27137	174,538	180	88	50	69.2 $\pm$ 19.2	65 $\pm$ 20.2
	LLVM-27747	173,840	117	79	68	87.8 $\pm$ 2.5	63.2 $\pm$ 2.2
	LLVM-31259	48,799	406	282	168	184.0 $\pm$ 51.2	114.4 $\pm$ 10.9
	GCC-59903	57,581	308	198	105	209.8 $\pm$ 72.1	166.4 $\pm$ 64.0
	GCC-60116	75,224	443	247	168	188.8 $\pm$ 52.4	127.6 $\pm$ 34.8
	GCC-61383	32,449	272	195	84	113.2 $\pm$ 13.6	105.2 $\pm$ 5.4
	GCC-61917	85,359	150	103	65	78.4 $\pm$ 10.6	73.4 $\pm$ 5.5
	GCC-64990	148,931	239	203	65	143.4 $\pm$ 58.0	119 $\pm$ 50.1
	GCC-65383	43,942	153	84	72	64.2 $\pm$ 2.7	64.2 $\pm$ 2.7
	GCC-66186	47,481	327	226	115	97.8 $\pm$ 17.9	94.2 $\pm$ 12.0
	GCC-66375	65,488	440	227	56	56.0 $\pm$ 5.1	56.0 $\pm$ 5.1
	GCC-70127	154,816	301	230	84	95.0 $\pm$ 3.8	73.6 $\pm$ 3.3
	GCC-70586	212,259	426	223	130	235.4 $\pm$ 30.2	156.8 $\pm$ 12.6
	GCC-71626	6,133	51	38	46	44.6 $\pm$ 3.4	36.6 $\pm$ 0.9
	Mean	94,487	247.8	157.4	85.7	105.8 $\pm$ 4.4	86.1 $\pm$ 2.9
Benchmark-Rust	Rust-44800	801	467	284	473	124.6 $\pm$ 32.4	118.6 $\pm$ 34.4
	Rust-66851	936	728	713	654	414.2 $\pm$ 273.9	331.2 $\pm$ 257.8
	Rust-69039	190	114	101	110	97.2 $\pm$ 8.0	90.8 $\pm$ 10.9
	Rust-77002	347	263	247	264	96.0 $\pm$ 27.9	96.0 $\pm$ 27.9
	Rust-77320	173	40	40	40	40.0 $\pm$ 0.0	39.0 $\pm$ 0.0
	Rust-77323	81	13	13	13	13.0 $\pm$ 0.0	13.0 $\pm$ 0.0
	Rust-77910	63	34	21	23	29.2 $\pm$ 2.7	21.0 $\pm$ 0.0
	Rust-77919	132	74	62	70	62.2 $\pm$ 14.3	58.2 $\pm$ 7.7
	Rust-78005	182	102	102	75	102.0 $\pm$ 0.0	102.0 $\pm$ 0.0
	Rust-78325	65	29	26	34	29.0 $\pm$ 0.0	26.0 $\pm$ 0.0
	Rust-78651	957	17	9	12	16.6 $\pm$ 0.5	11.0 $\pm$ 2.7
	Rust-78652	263	56	49	49	53.6 $\pm$ 2.2	49.0 $\pm$ 0.0
	Rust-78655	28	26	26	26	26.0 $\pm$ 0.0	26.0 $\pm$ 0.0
	Rust-78720	121	72	56	51	58.4 $\pm$ 2.1	56.6 $\pm$ 0.5
	Rust-91725	513	174	86	101	68.6 $\pm$ 57.6	55.2 $\pm$ 18.4
	Rust-99830	448	299	277	160	271.6 $\pm$ 12.5	230.2 $\pm$ 62.6
	Rust-111502	192	166	157	161	104.8 $\pm$ 7.2	103.6 $\pm$ 8.2
	Rust-112061	556	458	442	450	385.0 $\pm$ 32.6	380.4 $\pm$ 31.8
	Rust-112213	866	736	635	732	653.0 $\pm$ 34.0	618.8 $\pm$ 22.9
	Rust-112526	644	382	338	545	304.0 $\pm$ 30.3	283.2 $\pm$ 23.3
	Mean	378	212.5	184.2	202.2	147.5 $\pm$ 11.1	135.5 $\pm$ 10.1
Benchmark-JS	JS-1	244	52	41	41	25.6 $\pm$ 0.5	24.8 $\pm$ 1.6
	JS-2	112	51	41	40	34.0 $\pm$ 6.5	27.8 $\pm$ 3.5
	JS-3	125	57	41	47	51.6 $\pm$ 12.1	35.4 $\pm$ 7.7
	JS-4	185	65	35	47	33.4 $\pm$ 2.2	28.2 $\pm$ 6.9
	JS-5	178	66	38	52	41.6 $\pm$ 2.2	33.8 $\pm$ 5.5
	JS-6	152	57	30	45	20.0 $\pm$ 2.8	19.2 $\pm$ 2.7
	JS-7	144	46	38	38	34.0 $\pm$ 0.0	27.2 $\pm$ 6.3
	JS-8	121	55	47	45	40.6 $\pm$ 6.3	33.0 $\pm$ 17.4
	JS-9	87	50	30	32	23.8 $\pm$ 2.5	18.8 $\pm$ 5.2
	JS-10	63	56	41	43	34.8 $\pm$ 5.1	27.0 $\pm$ 6.3
	Mean	141	55.5	38.2	43.0	33.9 $\pm$ 1.6	27.5 $\pm$ 5.9

Further analysis into these benchmarks reveals that Vulcan and transformations in LPR are complementary on Benchmark-Rust. Vulcan demonstrates greater effectiveness in reducing programs of relatively smaller size, as highlighted by the average original size of 43 tokens across 9 programs, outperforming LPR. In contrast, the 15 programs where LPR is proved better than Vulcan have an average of 269 tokens, indicating its proficiency in reducing relatively larger programs. Our speculation is that Vulcan and LPR target different reduction opportunities. Vulcan performs identifier/sub-tree replacement and local exhaustive search. Such reducers, while lacking in semantic analysis, find reduction opportunities in a "brute-force" manner and are particularly effective at uncovering less obvious reduction opportunities. On the other hand, LPR employs more semantic and complex transformations, adeptly and systematically analyzing and reducing a complicated program step by step. Moreover, the results of LPR+Vulcan in the last column prove the complementary characteristic between LPR and Vulcan, which achieve the best in 16 out of the total 20 benchmarks.

Benchmark-JS. Programs in Benchmark-JS are much simpler and smaller than those in the previous two benchmark suites. Therefore, even Perses alone is capable of reducing the programs to only 55.5 tokens. Following this, Vulcan, LPR and LPR+Vulcan achieve 38.2, 33.9 and 27.5 tokens, further reducing the average results by 30.35%, 38.66% and 50.25%, separately. Similar to its performance on Rust, C-Reduce cannot outperform the aforementioned algorithms on JavaScript, due to its lack of employment of JavaScript’s semantics. The evaluation results also serve to demonstrate that the performance exhibited by LPR is not attributable to data leakage. These benchmarks were collected by the authors via fuzzing, and the optimal results remain inaccessible to the public, thereby precluding any possibility of LLMs memorizing them.

RQ1: LPR improves Perses by producing 51.33%, 14.87% and 38.66% smaller programs on three benchmarks. Moreover, LPR+Vulcan improves Vulcan, by 36.73%, 14.39% and 28.15%. On C language, LPR+Vulcan performs comparably to C-Reduce, a language-specific reducer for C language.

5.4.3 RQ2: Reduction Efficiency

In this research question, we measure the time elapsed when users employ each technique for program reduction. A shorter time indicates higher efficiency, and Table 5.2 shows the results. Since both Vulcan and LPR perform reduction on top of Perses’s results, it is impossible for these two algorithms to take less time than Perses. In addition, as a highly efficient tree-based reduction algorithm, Perses is generally faster than C-Reduce.

Therefore, we focus on the comparison among Vulcan, C-Reduce and LPR. It should be noted that the time measured in this research question cannot be directly used to compare the computational resources consumed by LPR with those consumed by other techniques. LPR is standing on the shoulders of giants, *i.e.*, LLMs, of which computations heavily rely on GPUs, and it is infeasible for us to measure the actual resource consumption of each query to OpenAI API. In contrast, Perses, Vulcan, and C-Reduce only leverage CPUs.

In Benchmark-C, compared to Vulcan and LPR, C-Reduce generally has a shorter reduction time. This is expected, as C-Reduce’s transformations are specifically designed for C languages, whereas Vulcan approaches the problem in a more unguided and brute-force manner, and LPR’s attempts are not specifically designed for C and rely more on the efficiency of LLMs. On average, LPR takes 1h:54m:54s, which is 10.77% shorter than 2h:08m:46s of Vulcan. However, in terms of the average percentage difference of each benchmark, LPR requires 45.83% more time compared to Vulcan in Benchmark-C. The main reason for such a result is that LPR is more efficient than Vulcan when the program is large and complex, as the transformations it performs are aware of the semantics and have a higher success rate in reducing the program. However, when the program is small and simple, Vulcan can finish quickly since its search space becomes considerably small, but for LPR, the time consumed by the LLM is not decreased significantly and becomes dominant, making LPR less efficient than Vulcan in these benchmarks. On Benchmark-Rust, the results indicate a similar trend, *i.e.*, LPR tends to be more efficient when reducing complex programs, while Vulcan is more efficient in small and simple benchmarks. If we only keep benchmarks where both tools take longer than one hour, LPR requires 4.15% and 21.69% less time compared to Vulcan.

In our analysis of Benchmark-Rust, we observed that both LPR and Vulcan can consume an extremely long time on certain benchmarks. For instance, Vulcan requires 20 hours for Rust-99830, while LPR takes a similar duration on Rust-66851 in a specific run. This extensive time consumption is often due to the frequent invocation of Perses, which only achieves marginal progress with each transformation. The prolonged duration of Perses is primarily attributed to the strict syntax of the Rust language. Considering that program reduction is an NP-complete problem, this inefficiency might be optimized in the future, whereas it cannot be completely eliminated.

After further in-depth analysis, another interesting fact emerges. On the three benchmark suites, the average time taken by LPR is 1.915, 1.839, and 0.174 hours, respectively. However, within these durations, the time spent waiting for the LLM responses accounts for 32.26%, 28.11%, and 72.99%. Additionally, we measure the efficiency from more perspectives. Take Benchmark-C as an example: the average expense to finish each benchmark amounts to \$0.42, with each benchmark requiring 41 queries on average, and every query consuming

Table 5.2: The reduction time (in the format of hh:mm:ss) of Perses, Vulcan, C-Reduce, LPR and LPR+Vulcan.

	Benchmark	Perses	Vulcan	C-Reduce	LPR	LPR+Vulcan
Benchmark-C	LLVM-22382	0:06:46	0:17:03	0:14:46	0:39:43 ± 0:17:13	0:44:33 ± 0:16:46
	LLVM-22704	0:33:58	0:38:03	0:22:38	0:48:53 ± 0:01:03	0:52:17 ± 0:01:23
	LLVM-23309	0:22:34	2:02:39	0:48:47	1:35:13 ± 0:13:36	2:05:40 ± 0:14:16
	LLVM-23353	0:10:31	0:15:05	0:13:36	0:25:23 ± 0:06:05	0:30:47 ± 0:06:33
	LLVM-25900	0:09:53	0:23:08	0:22:04	0:44:03 ± 0:07:56	0:54:33 ± 0:07:21
	LLVM-26760	0:21:54	0:34:23	0:32:25	0:54:40 ± 0:18:08	1:04:28 ± 0:16:51
	LLVM-27137	1:54:41	3:15:20	2:21:43	2:33:15 ± 0:09:08	3:13:24 ± 0:08:48
	LLVM-27747	0:13:32	0:28:02	0:25:30	0:32:04 ± 0:02:28	0:45:53 ± 0:03:17
	LLVM-31259	0:32:30	4:03:54	1:13:20	4:08:08 ± 0:34:26	5:01:39 ± 0:54:49
	GCC-59903	0:48:47	1:21:15	1:23:10	2:14:57 ± 0:45:14	2:44:43 ± 0:48:09
	GCC-60116	0:36:18	2:02:01	1:15:45	3:16:38 ± 0:31:53	4:25:39 ± 0:34:36
	GCC-61383	0:44:59	3:50:40	1:00:39	2:52:39 ± 0:30:41	4:36:47 ± 0:22:35
	GCC-61917	0:14:57	0:24:16	0:44:26	0:37:28 ± 0:07:37	0:43:30 ± 0:07:31
	GCC-64990	0:51:05	1:27:12	1:20:05	2:03:05 ± 0:31:38	2:22:37 ± 0:22:33
	GCC-65383	0:17:05	0:37:02	0:33:45	0:46:53 ± 0:05:25	0:59:14 ± 0:05:08
	GCC-66186	0:41:19	5:35:16	1:24:13	2:39:23 ± 0:19:26	4:05:34 ± 0:36:49
	GCC-66375	0:46:28	3:59:57	2:02:40	2:30:08 ± 0:10:24	3:06:41 ± 0:10:44
	GCC-70127	0:44:47	4:45:15	1:40:01	2:23:13 ± 0:20:36	3:24:03 ± 0:20:20
	GCC-70586	1:33:35	6:53:31	1:37:16	6:24:35 ± 1:26:57	10:36:26 ± 2:32:53
	GCC-71626	0:00:40	0:01:18	0:04:06	0:07:38 ± 0:01:27	0:08:11 ± 0:01:32
	Mean	0:35:19	2:08:46	0:59:03	1:54:54 ± 0:05:17	2:37:20 ± 0:08:05
Benchmark-Rust	Rust-44800	0:13:32	1:58:31	1:33:17	1:47:29 ± 0:31:37	2:15:39 ± 0:42:02
	Rust-66851	0:59:47	8:49:11	1:32:02	11:21:39 ± 9:56:51	16:43:40 ± 12:54:53
	Rust-69039	0:07:54	1:25:33	0:10:05	0:24:13 ± 0:05:33	0:39:22 ± 0:05:51
	Rust-77002	0:04:12	0:20:17	0:29:18	0:52:27 ± 0:15:54	0:58:21 ± 0:15:18
	Rust-77320	0:00:06	0:01:22	0:01:51	0:02:36 ± 0:00:32	0:04:05 ± 0:00:32
	Rust-77323	0:00:01	0:00:11	0:00:37	0:00:16 ± 0:00:03	0:00:27 ± 0:00:04
	Rust-77910	0:00:08	0:00:47	0:01:12	0:04:58 ± 0:01:34	0:05:44 ± 0:01:35
	Rust-77919	0:00:17	0:02:46	0:05:29	0:08:45 ± 0:05:08	0:11:18 ± 0:04:52
	Rust-78005	0:00:10	0:01:57	0:02:30	0:10:44 ± 0:01:32	0:12:55 ± 0:01:31
	Rust-78325	0:00:02	0:00:28	0:01:32	0:00:46 ± 0:00:35	0:01:19 ± 0:00:34
	Rust-78651	0:00:04	0:00:23	0:01:09	0:01:33 ± 0:00:33	0:02:02 ± 0:00:36
	Rust-78652	0:00:08	0:01:32	0:03:01	0:02:53 ± 0:02:02	0:04:38 ± 0:01:59
	Rust-78655	0:00:01	0:00:49	0:01:30	0:02:20 ± 0:00:31	0:03:14 ± 0:00:32
	Rust-78720	0:00:16	0:03:47	0:06:31	0:11:44 ± 0:04:47	0:13:52 ± 0:04:55
	Rust-91725	0:03:36	0:17:48	0:37:19	0:16:13 ± 0:03:55	0:23:29 ± 0:02:11
	Rust-99830	0:48:23	20:08:32	11:12:22	4:23:44 ± 1:02:03	24:28:10 ± 1:37:03
	Rust-111502	0:00:55	0:10:35	0:10:23	0:37:29 ± 0:11:08	0:44:39 ± 0:11:13
	Rust-112061	0:34:50	4:48:04	1:15:44	5:10:02 ± 3:12:51	8:01:03 ± 3:04:33
	Rust-112213	0:56:40	15:21:26	1:08:21	7:33:21 ± 2:44:52	17:07:36 ± 3:34:22
	Rust-112526	0:45:36	2:55:07	1:35:52	3:33:13 ± 1:21:21	4:52:26 ± 1:54:13
	Mean	0:13:50	2:49:27	1:00:30	1:50:19 ± 0:40:46	3:51:42 ± 0:44:16
Benchmark-JS	JS-1	0:01:29	0:29:19	0:06:59	0:11:47 ± 0:05:01	0:14:32 ± 0:05:00
	JS-2	0:02:15	0:17:10	0:11:47	0:14:36 ± 0:01:12	0:29:51 ± 0:04:21
	JS-3	0:01:29	0:26:35	0:06:02	0:09:00 ± 0:00:14	0:29:30 ± 0:07:58
	JS-4	0:00:19	0:15:43	0:01:44	0:07:22 ± 0:02:35	0:37:07 ± 0:03:57
	JS-5	0:00:14	0:04:25	0:01:39	0:10:32 ± 0:04:27	0:22:49 ± 0:04:32
	JS-6	0:02:59	0:23:20	0:11:33	0:13:24 ± 0:07:49	0:18:43 ± 0:08:08
	JS-7	0:00:15	0:03:30	0:16:21	0:05:28 ± 0:01:06	0:10:52 ± 0:01:09
	JS-8	0:00:44	0:13:48	0:02:21	0:08:18 ± 0:02:20	0:13:04 ± 0:02:10
	JS-9	0:01:32	0:14:14	0:07:50	0:09:09 ± 0:02:43	0:15:14 ± 0:03:40
	JS-10	0:01:31	0:17:44	0:07:24	0:15:04 ± 0:04:13	0:32:32 ± 0:05:37
	Mean	0:01:17	0:16:35	0:07:22	0:10:28 ± 0:01:26	0:22:25 ± 0:02:33

approximately 39 seconds on average. This experiment involves invoking the OpenAI API, and might have been limited by high user traffic and few computational resources allocated. Considering the ongoing advancements in LLM technology, we believe that LPR’s efficiency will be substantially improved in the future.

RQ2: From the perspective of users, LPR is more efficient than Vulcan on more complex programs with longer processing time, while Vulcan reduces faster than LPR on simpler and shorter programs.

5.4.4 RQ3: Effectiveness of Each Transformation

To answer this question, we delve into the impact of each transformation on program size, alongside their potential to escape the local minimal program and unlock new reduction opportunities. Our in-depth analysis focuses on Benchmark-C because of its complexity of bug-triggering programs compared to other benchmark suites, ensuring enough room for each transformation to take effect.

For each transformation, we measure how the program size changes in each benchmark of Benchmark-C after a specific transformation is performed, and plot the size changes into a box-plot and red dots, as shown in Fig. 5.4. Additionally, since each transformation is immediately followed by an invocation of Perses, we also monitor the cumulative size changes resulting from both the transformation and its subsequent Perses reduction. These changes are depicted by the blue dots in the second box-plot of each subplot in Fig. 5.4. Given that a transformation is performed in every iteration, we compute the average size change by dividing the total sum of size changes for that transformation by the number of iterations. This approach enables us to thoroughly comprehend how each transformation influences program size and assess its capacity to provide further reduction opportunities to Perses.

According to size changes induced by each transformation alone, *i.e.*, the left box-plot with red dots, we can find two trends. First, *Function Inlining*, *Data Type Elimination* and *Variable Elimination* are more likely to reduce the program size by themselves, with an average size change -13.2, -4.7 and -4.3, separately. However, for the rest two transformations, *i.e.*, *Loop Unrolling* and *Data Type Simplification*, most of the program sizes increase instead, with an average of 14.9 and 1.3 respectively. This is expected, as such transformations will generally transform the program into a larger one. *Loop Unrolling*, disassembles a loop into repeated lines of code, and leads to a size increase temporarily. *Data Type Simplification* can dismantle a variable in a complex data type, *e.g.*, structure, into a list of members in primary data types, which may require more tokens to declare and initialize each variable.

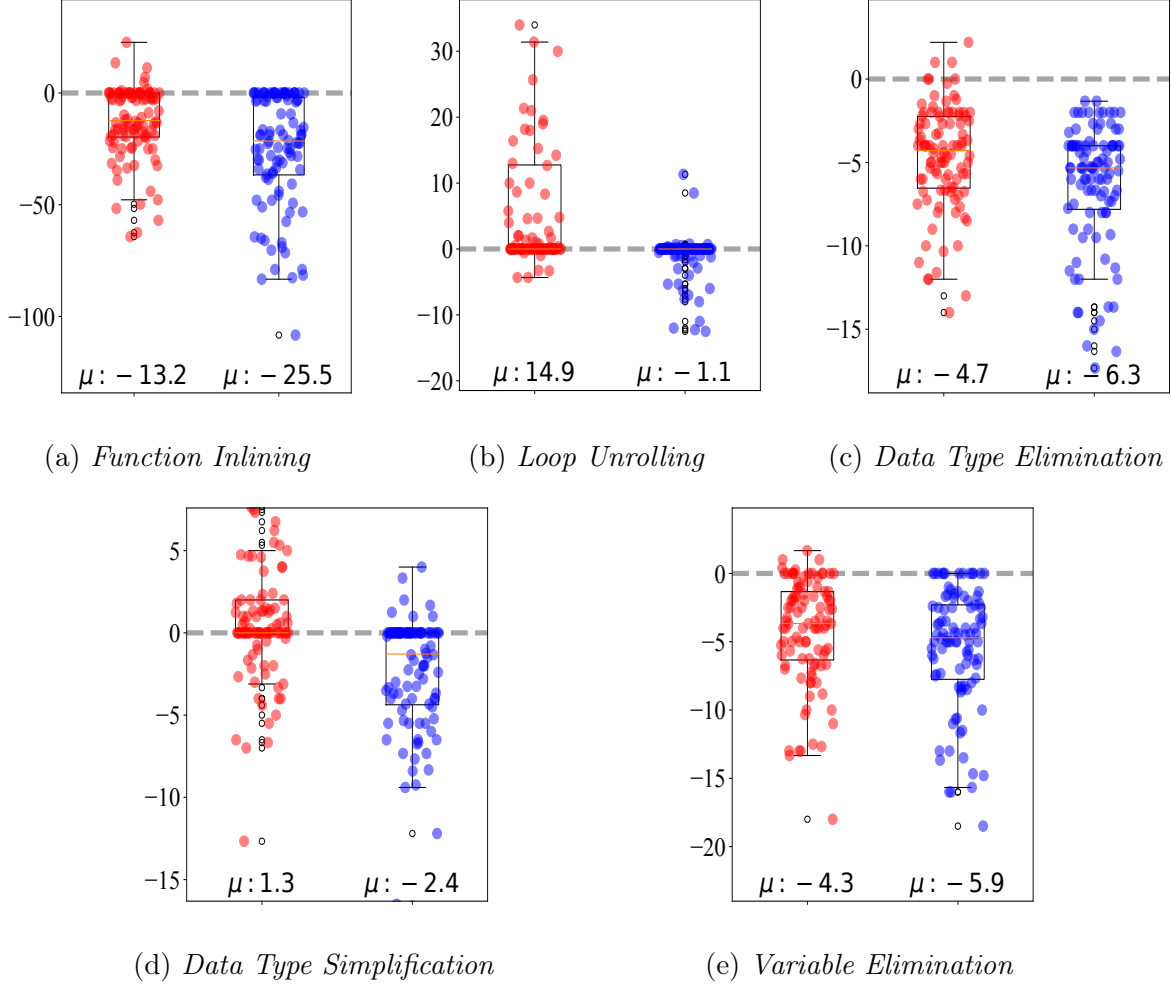


Figure 5.4: Program size changes induced by each transformation on benchmarks of Benchmark-C. In each subplot, the left box-plot and red dots represent how the size of each program changes before and after executing the transformation. The right box-plot and blue dots represent the size change of each benchmark after executing the transformation and the follow-up Perses reduction. There are a total of 20 benchmarks in Benchmark-C, and each experiment is repeated 5 times. Therefore, we draw 100 data points on each boxplot.

For size changes induced by both a transformation and the subsequent execution of Perses, *i.e.*, the right box-plot with blue dots, all of the proposed transformations result in size decreases. The fact that the right box-plot is generally lower than the left one indicates that Perses often further removes tokens after the transformation is applied. For *Loop Unrolling* and *Data Type Simplification*, even though they usually introduce more tokens to the program, they expose reduction opportunities for the following execution of Perses, and eventually result in a smaller program.

To further understand the impact of each transformation on the entirety of Benchmark-C, we provide a detailed analysis of their contributions. Specifically, on the 5 repeated experiments on 20 benchmarks, we calculate the average size reduction brought about by each transformation across all these 100 runs by summing up the size decreases attributed to the transformation in each benchmark and then computing the mean value, as illustrated in Fig. 5.5. Additionally, we quantify the prevalence of each transformation by counting the number of benchmarks in which it induces a size decrease, as demonstrated in Fig. 5.6. The above evaluations together offer a comprehensive view of how each transformation influences program sizes and how frequently they take effect within Benchmark-C.

From Fig. 5.5, it is evident that every transformation contributes to further size reduction in Benchmark-C. Specifically, while *Function Inlining* is responsible for a reduction of 88.2 tokens on average, contributing 62.12% to the overall decrease. *Loop Unrolling* shows a minimal effect, contributing only 4.8 tokens, 3.35% to the overall decrease. This highlights the varying degrees of influence each transformation has on program size. Further insights from Fig. 5.6 reveal that *Data Type Elimination* affects all benchmarks, likely due to the ubiquitous presence of `typedef` across all benchmarks. In contrast, *Loop Unrolling* is the least prevalent transformation, affecting merely 6 on average out of 20 benchmarks. This outcome is expected, considering that not all programs involve loop structures, and not every loop is irrelevant to the compiler bugs. In addition, the relatively higher standard deviations observed in the *Data Type Simplification* and *Loop Unrolling* suggest that these transformations present greater challenges for the LLM to effectively handle.

RQ3: In Benchmark-C, all proposed transformations contribute to the further reduction by either shrinking the programs directly or providing reduction opportunities to Perses.

5.5 Discussion

In this section, we discuss the effectiveness of multi-level prompts the performance of the LLM under different temperatures, and the failures in LPR.

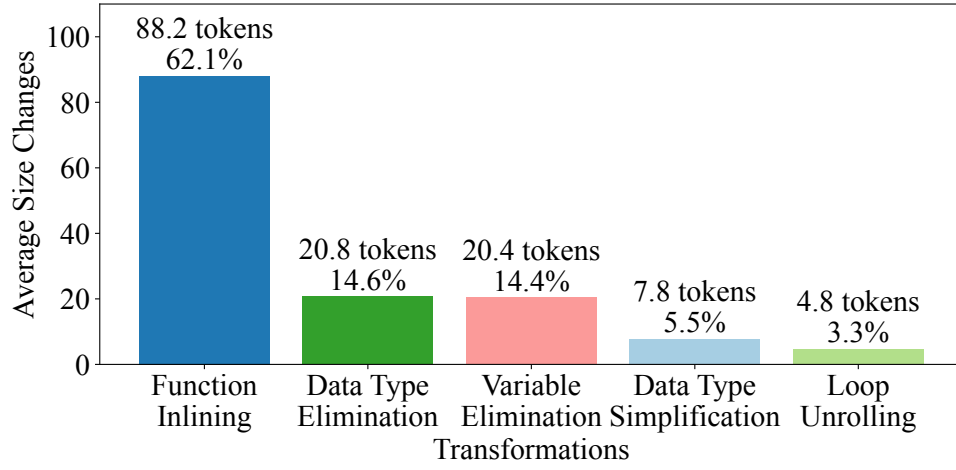


Figure 5.5: Average size decrease and its percentage induced by each transformation within Benchmark-C.

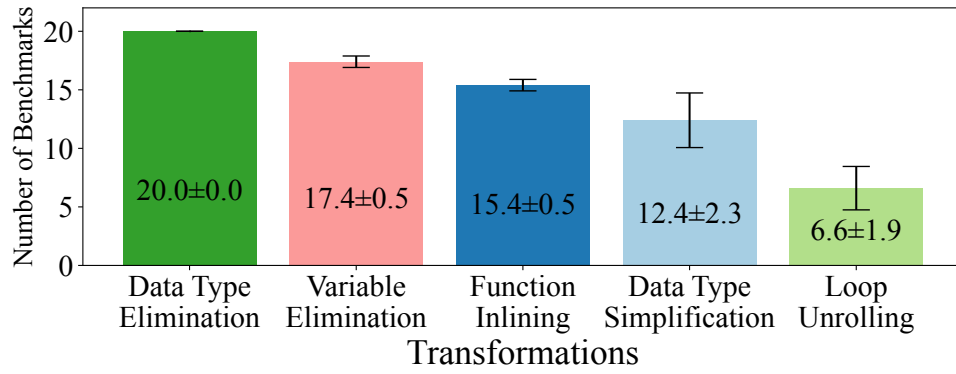


Figure 5.6: The number of benchmarks impacted by each transformation within Benchmark-C.

5.5.1 The Effectiveness of Multi-level Prompt

To validate the effectiveness of our proposed multi-level prompt, we design the corresponding single-level prompt and compare its effectiveness against the multi-level prompt. In detail, different from multi-level prompts, single-level prompts merge the *primaryQuestion* and *followupQuestion* into a single prompt, *e.g.*, “Given the following program { PROGRAM }, identify one function that can be inlined, and inline it.” for *Function Inlining*.

In 5 repeated experiments on Benchmark-C, the single-level prompting approach results in 155.0 ± 8.7 tokens, which is far less effective than results from the multi-level prompting approach, *i.e.*, 105.8 ± 4.4 tokens. Our explanation is that, even though such a single-level prompt is more compact, it makes LLMs less concentrated on a specific target, and thus LLMs may omit some targets.

5.5.2 The Impact of Temperature

In our experiments, we consistently set the temperature parameter of LLMs to 1.0. To measure how this parameter affects the performance, we rerun experiments under multiple temperatures, *i.e.*, 0.75, 0.5, 0.25, 0. Note that a higher temperature instructs the LLM to generate more creative and diverse results. Due to limited resources and time, we evaluate on 10 benchmarks in Benchmark-C with the fastest completion times under the default configuration.

Table 5.3: The impact of temperature on reduction sizes.

	Perses	Vulcan	LPR				
			$t = 1$	$t = 0.75$	$t = 0.5$	$t = 0.25$	$t = 0$
Mean	161.4	102.8	73.2 ± 3.0	72.2 ± 5.0	69.5 ± 1.0	71.1 ± 5.9	90.3 ± 3.8

As shown in Table 5.3, the performances under most of the temperatures are similar, while the exception is $t=0$, with the average size worse than others. According to the documentation of temperature [3], $t=0$ will actually use a small threshold above 0. Our speculation is that a low temperature restrains the diversity of outputs, impeding LPR exploring local minima in different runs, which is helpful in program reduction tasks.

Given that program reduction is an NP-complete problem, the randomness of LLMs has its advantages and drawbacks. Take the results of RQ1 in Table 5.1 as an example. On the one hand, randomness allows LPR to explore more distinct local minima, and

sometimes generates smaller programs than C-Reduce in five repeated experiments on each benchmark. On the other hand, the randomness of LLMs introduces variability. While the standard deviation remains below 10 in most benchmarks, it may increase in certain benchmarks, such as > 60 in GCC-59903. This variation can be attributed to the high complexity of the given program. In such scenarios, LLMs might not consistently execute accurate transformations, resulting in a range of local minima and divergent results.

5.5.3 Failures in LPR

In LPR, the generation of variants failing to pass the property test is common and acceptable. There are two scenarios where LPR fails to produce variants passing the property test. First, the LLM is generally non-deterministic, it may fail to perform the correct transformation when the program is complex. Therefore, multiple responses are requested in each LLM query to mitigate the impact of potential failures. In another scenario, the transformation may eliminate the bug-triggering pattern, e.g., the compiler crashes on a function call, but the transformation inlines this function. Even if such a transformation is semantically correct, the property test will still fail. When the transformed program is bound to fail, LPR will proceed with the next step using the original program.

Failures indeed lead to more property tests before making any progress, but they are inevitable, as program reduction is a trial-and-error process. Even without LLM, property checks issued by Perses and Vulcan usually have a failure rate of around 90%. As shown in Table 5.1 and Table 5.2, LPR is effective and efficient enough, illustrating that failures can be mitigated via multiple responses.

5.5.4 Threats to Validity

In this section, we discuss potential factors that may undermine the validity of our experimental results.

Threats to Internal Validity

The main internal threat comes from the potential data leakage problem. That is, do LLMs provide reasonable transformation through step-by-step analysis, or just simply memorize the minimal programs for the benchmark suites, which may be publicly available on the internet?

We mitigate this threat from several perspectives. First, program reduction is a task involving programs distinct from those used in program repair or program synthesis tasks. For instance, LLMs can learn from large datasets about code generation. On the contrary, programs in program reduction tasks are generally large, complex, and most importantly, randomly generated to trigger compiler bugs, with no other specific purpose. Their random and chaotic characteristics make it highly unlikely for LLMs to memorize such disorganized content, thus reducing the risk of data leakage. Moreover, even in scenarios where the LLMs might coincidentally memorize certain minimal programs, it is improbable for it to link a random-looking, featureless code snippet with a specific memorized program, especially when no explicit bug ID is provided. Furthermore, Benchmark-JS, one of the benchmark suites used, was created using JIT fuzzing tools by the authors and is not publicly accessible. This exclusivity ensures that the LLMs’ performance on these benchmarks reflects their ability to handle unseen and novel programs, thereby showcasing their effectiveness in managing new challenges without relying on memorized data. This approach significantly mitigates the risk of data leakage and demonstrates the capacity of LLMs for genuine problem-solving and analysis.

Threats to External Validity

One threat to external validity is the generalizability of LPR across languages. Although the approach of LPR is language-agnostic, LLMs used by LPR may have limited knowledge of certain languages, which may affect the performance of LPR. To mitigate this threat, we evaluate LPR on three prevalent programming languages, namely, C, Rust and JavaScript. The evaluation results demonstrated the generalizability of LPR on diverse popular programming languages. For languages that are not familiar to LLMs, LPR may still produce reasonable results, since it is shown that LLMs like Codex still perform well in less popular languages like Lua (0.2% in Github) [19]. As for a completely new language that LLMs cannot recognize and process, a possible solution is to incorporate the prompts in LPR with few-shot prompting [4, 78], so that LLMs learn how to recognize and process a new language from the given examples. Further exploration of this approach will be left as future work.

An additional threat is the applicability of our approach across different LLMs. To mitigate this threat, on the same benchmarks used in § 5.5.2, we repeat the experiments with CodeLlama, another widely used LLM family. As illustrated in Table 5.4, the experimental results on two models of CodeLlama, *i.e.*, CodeLlama-13b and CodeLlama-34b, show no distinct differences from results on ChatGPT. Furthermore, as LLMs continue to evolve, we anticipate improvements in both quality and time of code processing.

Table 5.4: The reduction sizes of LPR using various LLMs.

	Perses	Vulcan	LPR with gpt-3.5-turbo-0613	LPR with CodeLlama 13b	34b
Mean	161.4	102.8	73.2 ± 3.0	73.4 ± 3.5	71.3 ± 4.9

5.6 Related Work

We introduce related work in two topics: program reduction and LLMs for software engineering.

5.6.1 Program Reduction

ddmin [145] initiated the research topic of program reduction. It treats the input as a list of elements, and consistently splits the list into halves. Then it iteratively attempts to reduce the input list by exploring subsets and their complements at varying levels of granularity, transitioning from coarse to fine. Hierarchical Delta Debugging [86], short for HDD, parses the program input into a parsing tree, and performs ddmin on each level of the tree structure. Perses [109] avoids the generation of syntactically invalid program variants during reduction by formal syntax transformation. Vulcan, further pushes the limit of Perses via identifier/sub-tree replacement and local exhaustive search [143]. RCC is a compact refreshable caching scheme to speed up program reduction [118]. All the above works are not customized for certain languages, though having high generality, they lack semantic knowledge of a certain language for further reduction.

Besides, some tools are specifically designed for certain languages. C-Reduce [99], incorporating various transformation passes for features in C, is the most effective reducer on this language. J-Reduce [55, 56] is a tool for Java bytecode reduction, it reformulates the bytecode reduction into dependency graph simplification. ddSMT [92] is designed for reducing programs in SMT-LIBv2 format. All these works leverage language features to reduce more effectively than language-generic tools.

Distinct from prior work, LPR synergistically combines LLMs and language-generic reduction tools to harness the advantages of both. Language-generic reducers stand out for their remarkable generality across multiple languages, while LLMs excel in further refining the programs with the domain knowledge of certain languages learned from large training sets. Language-specific tools typically demand considerable human effort to design and

implement feature-related transformations for reduction, while LPR requires only a few lines of natural language prompts, substantially reducing the effort involved.

5.6.2 LLMs for Software Engineering

Large Language Models (LLMs) have proved their remarkable capability of undertaking multiple text-processing tasks, including source code-related works. Recent works focus on applying LLMs to facilitate software engineering tasks, or assessing the effectiveness, potential and limitations of LLMs on software development and maintenance. Some research [53, 139, 138, 137] focuses on empirically applying LLMs to automatic program repair (APR). Huang *et al.* [53] performed an empirical study on improvement brought by model fine-tuning in APR. Xia *et al.* [138] thoroughly evaluated 9 state-of-the-art LLMs across multiple datasets and programming languages, and demonstrated that directly applying LLMs has already outperformed all existing APR techniques. Additionally, some works focus on LLMs' performance *w.r.t.* code completion, generation and fuzzing [77, 156, 113, 27], by leveraging the code analysis and generation ability of LLMs.

Similar to these studies, our approach LPR leverages LLMs for a software engineering task, *i.e.*, program reduction. LPR harnesses the comprehension and generation capabilities of LLMs to refine the results of program reduction. However, our work distinguishes itself in the nature of the programs processed by LLMs. In related research, programs are typically logical and goal-oriented, often designed to fulfill a specific purpose. In contrast, the programs involved in our program reduction task are random, chaotic, and lack a clear objective. Consequently, our research sheds light on the performance of LLMs when dealing with programs that do not have an obvious purpose.

5.7 Conclusion

This chapter proposes LLM-aided program reduction (LPR), which is the first approach that leverages LLMs for the program reduction task to the best of our knowledge. By combining the strength of LLMs and existing language-generic program reduction techniques, LPR can perform language-specific transformations to effectively reduce the program while being language-generic (*i.e.*, can be easily applied to a wide range of languages). The evaluation shows that in 50 benchmarks across three programming languages. LPR outperforms Vulcan by producing 24.93%, 4.47%, and 11.71% smaller programs on C, Rust, and JavaScript, respectively. Meanwhile, The evaluation also demonstrates that LPR complements Vulcan

to some extent. By reducing the outputs of LPR with Vulcan, we attained results that have similar sizes to those of C-Reduce in Benchmark-C. In terms of efficiency perceived by users, LPR excels in reducing complex programs, and takes 10.77%, 34.88%, 36.96% less time than Vulcan to finish all the benchmarks, respectively.

Chapter 6

Conclusion and Future Work

In this chapter, we summarize this thesis and discuss some research topics for future work.

6.1 Summary

This thesis improves program reduction by proposing novel and effective techniques, conducting theoretical analysis, and providing insightful suggestions. It enhances the versatility, insights, and efficacy of program reduction through comprehensive exploration of these three aspects. To support this thesis statement, we present three studies, each targeting one of these core aspects in depth. Each study also aligns with a specific level of reduction: the second study focuses on the foundational list-based reduction, the first study explores the tree-based reduction, and the third study advances the semantics-aware transformation using large language models.

The first study in § 3 enhances the versatility of program reduction with PPR, a novel approach that focuses on reducing pairs of programs rather than individual ones. Operating at the tree-based level, PPR leverages parse trees to compute and minimize structural differences between program variants. By targeting pairs, PPR preserves the relational properties between programs, enabling a deeper understanding of the differences between bug-triggering variants and their non-bug-triggering counterparts. This method significantly improves the insights provided by the reduction process, offering a more comprehensive view of the bug-related components. The evaluation results demonstrate that PPR performs comparably to traditional single-program reduction methods and provides additional insights through the comparison of program pairs.

The second study in § 4 delves into deepening the insights into the state-of-the-art ProbDD algorithm, which operates at the list-based level. This study combines theoretical analysis with empirical evaluation to uncover the underlying mechanisms that contribute to ProbDD’s superior performance in program reduction. By simplifying the algorithm’s probabilistic model and validating these insights through experiments, the study reveals key factors driving its success. Furthermore, it offers practical strategies to optimize the performance of the ddmin algorithm and its variants. The introduction of CDD, a simplified version of ProbDD, serves to make the underlying processes more comprehensible while achieving similar performance, ultimately helping researchers and users apply these techniques more effectively.

The third study in § 5 focuses on improving the efficacy of program reduction by integrating large language models (LLMs) into the process. Aimed at the semantics-aware level, this study introduces LPR, which combines language-generic reducers with the semantic-aware capabilities of LLMs. By leveraging the strengths of both approaches, LPR is able to perform language-specific transformations while maintaining the generality of traditional reducers. The iterative process of reducing programs and applying language-specific optimizations enhances both the efficiency and effectiveness of the reduction process. Comprehensive evaluations across multiple benchmarks in C, Rust, and JavaScript demonstrate the superiority of LPR in terms of both reduced program size and faster processing times when compared to Vulcan.

6.2 Future Work

In program reduction, there are still directions worth exploring. First, most studies measure the size of the reduced program by counting the number of tokens, using this as a criterion for evaluating the effectiveness of the reduction. However, some approaches, such as T-Rec and C-Reduce, incorporate symbol renaming techniques to shorten variable names, further reducing the byte size of the program. From the perspective of compiler developers, a higher token count may not necessarily indicate a less effective reduction. This is because developers often lower the source programs to more low-level representations, such as LLVM-IR [63], where the size of the source programs may not directly reflect the complexity of the low-level representations. Therefore, rethinking the way to measure the effectiveness of program reduction approaches could lead to a more scientific evaluation of program reduction studies.

Besides, with the emergence of LLMs, particularly those fine-tuned for code optimizations [101, 72, 26], there is significant potential to further enhance the workflow of LPR.

For instance, LPR currently relies on user-defined prompts, but this could be optimized by allowing LLMs to autonomously determine the optimizations. In this way, LPR could identify potentially beneficial transformations while avoiding unnecessary ones, leading to enhanced performance. Future work may explore adaptive prompting strategies, where LLMs learn to select or generate optimizations based on the program context. Such advancements would not only improve the efficiency and generality of LPR but also push the boundary of semantic-aware reduction by enabling more intelligent and automated transformations.

References

- [1] 2023. *OpenAI API*. Retrieved 2023-11-20 from <https://platform.openai.com/docs/overview>
- [2] 2023. *OpenAI API: N*. Retrieved 2023-11-20 from <https://platform.openai.com/docs/api-reference/chat/create#chat-create-n>
- [3] 2023. *OpenAI API: Temperature*. Retrieved 2023-11-20 from <https://platform.openai.com/docs/api-reference/chat/create#chat-create-temperature>
- [4] Toufique Ahmed and Premkumar Devanbu. 2022. Few-shot training llms for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM international conference on automated software engineering*. 1–5. <https://doi.org/10.1145/3551349.3559555>
- [5] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. 2007. *Compilers: principles, techniques, & tools*. Pearson Education India. <https://doi.org/doi/10.5555/1177220>
- [6] Mohannad Alhanahnah, Rithik Jain, Vaibhav Rastogi, Somesh Jha, and Thomas Reps. 2022. Lightweight, multi-stage, compiler-assisted application specialization. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 251–269. <https://doi.org/csd1/proceedings-article/eurosp/2022/161400a251/1ErpBUm45c4>
- [7] ANTLR. 2012. *grammars-v4*. Retrieved 2023-11-23 from <https://github.com/antlr/grammars-v4>
- [8] Cyrille Artho. 2011. Iterative delta debugging. *International Journal on Software Tools for Technology Transfer* 13 (2011), 223–246. https://doi.org/10.1007/978-3-642-01702-5_13

- [9] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *NDSS*. <https://doi.org/10.14722/ndss.2019.23412>
- [10] ASF Bugzilla. 2001. *ASF Bugzilla: Bug Writing Guidelines*. <https://bz.apache.org/bugzilla/page.cgi?id=bug-writing.html> Accessed: 2023-04-30.
- [11] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. 2022. JIT-picking: Differential fuzzing of JavaScript engines. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 351–364. <https://doi.org/doi/10.1145/3548606.3560624>
- [12] Rudy Braquehais and Colin Runciman. 2017. Extrapolate: generalizing counterexamples of functional test properties. In *Proceedings of the 29th Symposium on the Implementation and Application of Functional Programming Languages*. 1–11. <https://doi.org/10.1145/3205368.3205371>
- [13] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901. <https://doi.org/10.5555/3495724.3495883>
- [14] Bobby R Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. 2020. Jshrink: In-depth investigation into debloating modern java applications. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 135–146. <https://doi.org/doi/10.1145/3368089.3409738>
- [15] Bugzilla. 2001. *Bugzilla: Reporting a New Bug*. <https://bugzilla.readthedocs.io/en/5.2/using/filing.html#reporting-a-new-bug> Accessed: 2023-04-30.
- [16] GCC Bugzilla. 2015. *Bug 66186*. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=66186 Accessed: 2023-04-30.
- [17] GCC Bugzilla. 2015. *Bug 66691*. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=66691 Accessed: 2023-04-30.
- [18] LLVM Bugzilla. 2014. *Bug 21467 - clang hangs on valid code at -Os and above on x86_64-linux-gnu*. https://bugs.llvm.org/show_bug.cgi?id=21467 Accessed: 2023-04-30.

- [19] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2022. Multipl-e: A scalable and extensible approach to benchmarking neural code generation. *arXiv preprint arXiv:2208.08227* (2022). <https://doi.org/10.48550/arXiv.2208.08227>
- [20] Stefanos Chaliasos, Thodoris Sotiropoulos, Diomidis Spinellis, Arthur Gervais, Benjamin Livshits, and Dimitris Mitropoulos. 2022. Finding typing compiler bugs. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 183–198. <https://doi.org/doi/10.1145/3519939.3523427>
- [21] Junjie Chen, Jiaqi Han, Peiyi Sun, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. Compiler bug isolation via effective witness test program generation. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 223–234. <https://doi.org/10.1145/3338906.3338957>
- [22] Junjie Chen, Haoyang Ma, and Lingming Zhang. 2020. Enhanced compiler bug isolation via memoized search. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 78–89. <https://doi.org/10.1145/3324884.3416570>
- [23] Ophelia C Chesley, Xiaoxia Ren, and Barbara G Ryder. 2005. Crisp: A debugging tool for Java programs. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, 401–410. <https://doi.org/document/1510135>
- [24] Arpit Christi, Alex Groce, and Rahul Gopinath. 2017. Resource adaptation via test-based software minimization. In *2017 IEEE 11th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. IEEE, 61–70. <https://doi.org/document/8064030>
- [25] James Cogan. 2017. *The patience diff algorithm*. <https://blog.jcogan.com/2017/09/19/the-patience-diff-algorithm> Accessed: 2023-04-30.
- [26] Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. 2024. Meta large language model compiler: Foundation models of compiler optimization. *arXiv preprint arXiv:2407.02524* (2024).
- [27] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning

- libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*. 423–435. <https://doi.org/10.1145/3597926.3598067>
- [28] Alastair Donaldson and David MacIver. 2021. *Test Case Reduction: Beyond Bugs*. Retrieved May 29, 2023 from <https://blog.sigplan.org/2021/05/25/test-case-reduction-beyond-bugs>
- [29] Alastair F Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated testing of graphics shader compilers. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29. <https://doi.org/10.1145/3133917>
- [30] Alastair F Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpiński. 2020. *Spirv-fuzz*. <https://github.com/google/graphicsfuzz/blob/master/docs/finding-a-vulkan-driver-bug-using-spirv-fuzz.md> Accessed: 2023-04-30.
- [31] Alastair F Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpiński. 2021. Test-case reduction and deduplication almost for free with transformation-based compiler testing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1017–1032. <https://doi.org/10.1145/3453483.3454092>
- [32] Georg Dotzler and Michael Philippsen. 2016. Move-optimized source code tree differencing. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. 660–671. <https://doi.org/10.1145/2970276.2970315>
- [33] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. 313–324. <https://doi.org/10.1145/2642937.2642982>
- [34] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 31–53. <https://doi.org/10.1109/ICSE-FoSE59343.2023.00008>
- [35] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *2023 IEEE/ACM*

- 45th International Conference on Software Engineering (ICSE). IEEE, 1469–1481. <https://doi.org/10.1109/ICSE48619.2023.00128>
- [36] Beat Fluri, Michael Wursch, Martin Pinzger, and Harald Gall. 2007. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering* 33, 11 (2007), 725–743. <https://doi.org/10.1109/TSE.2007.70731>
 - [37] GCC. 2020. *A Guide to Testcase Reduction*. https://gcc.gnu.org/wiki/A_guide_to_testcase_reduction Accessed: 2023-04-30.
 - [38] Masoud Ghaffarinia and Kevin W Hamlen. 2019. Binary control-flow trimming. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1009–1022. <https://doi.org/doi/10.1145/3319535.3345665>
 - [39] Golnaz Gharachorlu and Nick Sumner. 2018. Avoiding the familiar to speed up test case reduction. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 426–437. <https://doi.org/document/8424994>
 - [40] Golnaz Gharachorlu and Nick Sumner. 2019. PARDIS: Priority Aware Test Case Reduction. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 409–426. https://doi.org/10.1007/978-3-030-16722-6_24
 - [41] Rahul Gopinath, Alexander Kampmann, Nikolas Havrikov, Ezekiel O Soremekun, and Andreas Zeller. 2020. Abstracting failure-inducing inputs. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 237–248. <https://doi.org/10.1145/3395363.3397349>
 - [42] Alex Groce, Mohammed Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. 2014. Cause reduction for quick testing. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 243–252. <https://doi.org/document/6823886>
 - [43] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. 2016. Cause reduction: delta debugging, even without bugs. *Software Testing, Verification and Reliability* 26, 1 (2016), 40–68. <https://doi.org/doi/10.1002/stvr.1574>
 - [44] Qiuhan Gu. 2023. LLM-Based Code Generation Method for Golang Compiler Testing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference*

and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 2201–2203. <https://doi.org/10.1145/3611643.3617850>

- [45] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective program debloating via reinforcement learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 380–394. <https://doi.org/10.1145/3243734.3243838>
- [46] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. 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, 380–394. <https://doi.org/10.1145/3243734.3243838>
- [47] Satia Herfert, Jibesh Patra, and Michael Pradel. 2017. Automatically reducing tree-structured test inputs. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 861–871. <https://doi.org/document/8115697>
- [48] Renáta Hodován and Ákos Kiss. 2016. Modernizing hierarchical delta debugging. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*. 31–37. <https://doi.org/doi/10.1145/2994291.2994296>
- [49] Renáta Hodován and Ákos Kiss. 2016. Practical Improvements to the Minimizing Delta Debugging Algorithm.. In *ICSOFTEA*. 241–248. <https://doi.org/Link.aspx?doi=10.5220/0005988602410248>
- [50] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2017. Coarse hierarchical delta debugging. In *2017 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 194–203. <https://doi.org/document/8094421>
- [51] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*. 445–458. <https://doi.org/10.5555/2362793.2362831>
- [52] Michael Howard and David LeBlanc. 2003. *Writing secure code*. Pearson Education.
- [53] Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023. An Empirical Study on Fine-Tuning Large Language Models

- of Code for Automated Program Repair. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 1162–1174. <https://doi.org/10.1109/ASE56229.2023.00181>
- [54] Naman Jain, Skanda Vaidyanath, Arun Shankar Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram K. Rajamani, and Rahul Sharma. 2022. Jigsaw: Large Language Models meet Program Synthesis. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1219–1231. <https://doi.org/10.1145/3510003.3510203>
 - [55] Christian Gram Kalhauge and Jens Palsberg. 2019. Binary reduction of dependency graphs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 556–566. <https://doi.org/10.1145/3338906.3338956>
 - [56] Christian Gram Kalhauge and Jens Palsberg. 2021. Logical bytecode reduction. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1003–1016. <https://doi.org/10.1145/3453483.3454091>
 - [57] Akos Kiss. 2020. Generalizing the split factor of the minimizing delta debugging algorithm. *IEEE Access* 8 (2020), 219837–219846. <https://doi.org/document/9285235>
 - [58] Ákos Kiss, Renáta Hodován, and Tibor Gyimóthy. 2018. HDDr: a recursive variant of the hierarchical delta debugging algorithm. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. 16–22. <https://doi.org/doi/10.1145/3278186.3278189>
 - [59] Akos Kiss, Renáta Hodován, and Daniel Vince. 2016. *Picire*. <https://github.com/renatahodovan/picire> Accessed: 2023-04-30.
 - [60] Akos Kiss, Renáta Hodován, and Daniel Vince. 2016. *Picireny*. <https://github.com/renatahodovan/picireny> Accessed: 2023-04-30.
 - [61] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. 2019. Configuration-driven software debloating. In *Proceedings of the 12th European Workshop on Systems Security*. 1–6. <https://doi.org/doi/10.1145/3301417.3312501>
 - [62] Gray Kwong, Jesse Ruderman, and Jesse Schwartzentuber. 2022. *Lithium algorithm*. Retrieved 2022-12-24 from <https://github.com/MozillaSecurity/lithium>

- [63] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004*. IEEE, 75–86.
- [64] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices* 49, 6 (2014), 216–226. <https://doi.org/10.1145/2594291.2594334>
- [65] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. *ACM SIGPLAN Notices* 50, 10 (2015), 386–399. <https://doi.org/10.1145/2858965.2814319>
- [66] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Randomized stress-testing of link-time optimizers. In *Proceedings of the 2015 international symposium on software testing and analysis*. 327–337. <https://doi.org/doi/10.1145/2771783.2771785>
- [67] Bastien Lecoœur, Hasan Mohsin, and Alastair F Donaldson. 2023. Program reconditioning: Avoiding undefined behaviour when finding and reducing compiler bugs. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1801–1825. <https://doi.org/doi/10.1145/3591294>
- [68] Bastien Lecoœur, Hasan Mohsin, and Alastair F. Donaldson. 2023. Program Reconditioning: Avoiding Undefined Behaviour When Finding and Reducing Compiler Bugs. *Proc. ACM Program. Lang.* 7, PLDI, Article 180 (jun 2023), 25 pages. <https://doi.org/10.1145/3591294>
- [69] Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 42–54. <https://doi.org/10.1145/1111037.1111042>
- [70] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. <https://doi.org/10.1145/1538788.1538814>
- [71] Junyi Li, Xiaoxue Cheng, Xin Zhao, Jian-Yun Nie, and Ji-Rong Wen. 2023. HaluEval: A Large-Scale Hallucination Evaluation Benchmark for Large Language Models. In *The 2023 Conference on Empirical Methods in Natural Language Processing*. <https://doi.org/10.48550/arXiv.2305.11747>

- [72] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).
- [73] Shuxin Li and Manuel Rigger. 2024. Finding XPath Bugs in XML Document Processors via Differential Testing. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12. <https://doi.org/doi/10.1145/3597503.3639208>
- [74] Shaohua Li, Theodoros Theodoridis, and Zhendong Su. 2024. Boosting compiler testing by injecting real-world code. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 223–245. <https://doi.org/doi/10.1145/3656386>
- [75] Tsz On Li, Wenxi Zong, Yibo Wang, Haoye Tian, Ying Wang, Shing-Chi Cheung, and Jeff Kramer. 2023. Nuances are the Key: Unlocking ChatGPT to Find Failure-Inducing Tests with Differential Prompting. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 14–26. <https://doi.org/doi/10.1109/ASE56229.2023.00089>
- [76] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F Donaldson. 2015. Many-core compiler fuzzing. *ACM SIGPLAN Notices* 50, 6 (2015), 65–76. <https://doi.org/doi/10.1145/2737924.2737986>
- [77] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2023), 21558–21572. <https://doi.org/doi/10.5555/3666122.3667065>
- [78] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *Comput. Surveys* 55, 9 (2023), 1–35. <https://doi.org/doi/10.1145/3560815>
- [79] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–25. <https://doi.org/doi/10.1145/1993498.1993532>
- [80] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2023. Fuzzing loop optimizations in compilers for C++ and data-parallel languages. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1826–1847.

- [81] LLVM. 2000. *LibTooling*. <https://clang.llvm.org/docs/LibTooling.html> Accessed: 2023-04-30.
- [82] LLVM. 2022. *How to submit an LLVM bug report*. <https://llvm.org/docs/HowToSubmitABug.html> Accessed: 2023-04-30.
- [83] David Majnemer. 2014. *InstCombine: Remove infinite loop caused by FoldOpIntoPhi*. Retrieved 2022-07-04 from <https://reviews.llvm.org/rG7e2b9882b147bf9c26faa7b04b56884ec444bd64>
- [84] Gregory Malecha, Ashish Gehani, and Natarajan Shankar. 2015. Automated software winnowing. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. 1504–1511. <https://doi.org/doi/10.1145/2695664.2695751>
- [85] Barton P Miller, Mengxiao Zhang, and Elisa R Heymann. 2020. The relevance of classic fuzz testing: Have we solved this one? *IEEE Transactions on Software Engineering* 48, 6 (2020), 2028–2039. <https://doi.org/doi/abs/10.1109/TSE.2020.3047766>
- [86] Ghassan Mishherghi and Zhendong Su. 2006. HDD: hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering*. 142–151. <https://doi.org/10.1145/1134285.1134307>
- [87] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. 2013. Compiler testing via a theory of sound optimisations in the C11/C++ 11 memory model. *ACM SIGPLAN Notices* 48, 6 (2013), 187–196. <https://doi.org/10.1145/2499370.2491967>
- [88] MozillaSecurity. 2022. *Using Lithium*. <https://github.com/MozillaSecurity/lithium> Accessed: 2023-04-30.
- [89] Eugene W Myers. 1986. AnO (ND) difference algorithm and its variations. *Algorithmica* 1, 1 (1986), 251–266.
- [90] Eriko Nagai, Hironobu Awazu, Nagisa Ishiura, and Naoya Takeda. 2012. Random testing of C compilers targeting arithmetic optimization. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2012)*. 48–53. <https://doi.org/10.1145/2499370.2491967>
- [91] Eriko Nagai, Atsushi Hashimoto, and Nagisa Ishiura. 2014. Reinforcing random testing of arithmetic optimization of C compilers by scaling up size and number of expressions. *IPSI Transactions on System LSI Design Methodology* 7 (2014), 91–100. <https://doi.org/10.2197/ipsjtsldm.7.91>

- [92] Aina Niemetz and Armin Biere. 2013. ddSMT: a delta debugger for the SMT-LIB v2 format. In *Proceedings of the 11th International Workshop on Satisfiability Modulo Theories, SMT*. 8–9.
- [93] OpenAI. 2023. *Six Strategies for Getting Better Results with GPT*. <https://platform.openai.com/docs/guides/prompt-engineering/six-strategies-for-getting-better-results> Accessed: November 15, 2023.
- [94] Mateusz Pawlik and Nikolaus Augsten. 2011. RTED: a robust algorithm for the tree edit distance. *arXiv preprint arXiv:1201.0230* (2011). <https://doi.org/10.14778/2095686.2095692>
- [95] Martin Pelikan, David E Goldberg, Erick Cantú-Paz, et al. 1999. BOA: The Bayesian optimization algorithm. In *Proceedings of the genetic and evolutionary computation conference GECCO-99*, Vol. 1. Citeseer, 525–532. <https://doi.org/doi/10.5555/2933923.2933973>
- [96] Lee Pike. 2014. SmartCheck: automatic and efficient counterexample reduction and generalization. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell*. 53–64. <https://doi.org/10.1145/2775050.2633365>
- [97] Chenxiong Qian, Hong Hu, Mansour Alharthi, Simon Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating.. In *USENIX Security Symposium*. 1733–1750. <https://doi.org/doi/10.5555/3361338.3361459>
- [98] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating software through piecewise compilation and loading. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 869–886. <https://doi.org/doi/10.5555/3277203.3277269>
- [99] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 335–346. <https://doi.org/10.1145/2254064.2254104>
- [100] John et al. Regehr. 2012. *C-Reduce*. Retrieved 2023-11-26 from <https://github.com/csmith-project/creduce>
- [101] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023.

Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

- [102] Fabian Ruffy, Tao Wang, and Anirudh Sivaraman. 2020. Gauntlet: Finding bugs in compilers for programmable packet processing. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 683–699.
- [103] Uri Shaham, Elad Segal, Maor Ivgi, Avia Efrat, Ori Yoran, Adi Haviv, Ankit Gupta, Wenhan Xiong, Mor Geva, Jonathan Berant, and Omer Levy. 2022. SCROLLS: Standardized CompaRison Over Long Language Sequences. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (Eds.). Association for Computational Linguistics, 12007–12021. <https://doi.org/10.18653/V1/2022.EMNLP-MAIN.823>
- [104] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. TRIMMER: application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 329–339. <https://doi.org/TRIMMER:applicationspecializationforcodedebloating>
- [105] Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H Chi, Nathanael Schärli, and Denny Zhou. 2023. Large language models can be easily distracted by irrelevant context. In *International Conference on Machine Learning*. PMLR, 31210–31227. <https://doi.org/10.5555/3618408.3619699>
- [106] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding and analyzing compiler warning defects. In *Proceedings of the 38th International Conference on Software Engineering*. 203–213.
- [107] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 849–863. <https://doi.org/10.1145/2983990.2984038>
- [108] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 294–305. <https://doi.org/10.1145/2931037.2931074>
- [109] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-guided program reduction. In *Proceedings of the 40th International*

Conference on Software Engineering. 361–371. <https://doi.org/10.1145/3180155.3180236>

- [110] Maolin Sun, Yibiao Yang, Yang Wang, Ming Wen, Haoxiang Jia, and Yuming Zhou. 2023. SMT Solver Validation Empowered by Large Pre-Trained Language Models. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 1288–1300. <https://doi.org/10.1109/ASE56229.2023.00180>
- [111] Yutian Tang, Hao Zhou, Xiapu Luo, Ting Chen, Haoyu Wang, Zhou Xu, and Yan Cai. 2021. Xdebloat: Towards automated feature-oriented app debloating. *IEEE Transactions on Software Engineering* 48, 11 (2021), 4501–4520. <https://doi.org/document/9573340>
- [112] Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. 2022. Finding missed optimizations through the lens of dead code elimination. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 697–709.
- [113] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. 2023. Is ChatGPT the Ultimate Programming Assistant—How far is it? *arXiv preprint arXiv:2304.11938* (2023). <https://doi.org/10.48550/arXiv.2304.11938>
- [114] Jia Le Tian, Mengxiao Zhang, Zhenyang Xu, Yongqiang Tian, Yiwen Dong, and Chengnian Sun. 2023. Ad Hoc Syntax-Guided Program Reduction. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (San Francisco, CA, USA, 3–9) (ESEC/FSE ’23)*. ACM, New York, NY, USA, 5. <https://doi.org/10.1145/3611643.3613101>
- [115] Jia Le Tian, Mengxiao Zhang, Zhenyang Xu, Yongqiang Tian, Yiwen Dong, and Chengnian Sun. 2023. Ad Hoc Syntax-Guided Program Reduction. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 2137–2141. <https://doi.org/10.1145/3611643.3613101>
- [116] Yongqiang Tian, Zhenyang Xu, Yiwen Dong, Chengnian Sun, and Shing-Chi Cheung. 2023. Revisiting the Evaluation of Deep Learning-Based Compiler Testing. In

Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, 19th-25th August 2023, Macao, SAR, China. ijcai.org, 4873–4882. <https://doi.org/10.24963/IJCAI.2023/542>

- [117] Yongqiang Tian, Xueyan Zhang, Yiwen Dong, Zhenyang Xu, Mengxiao Zhang, Yu Jiang, Shing-Chi Cheung, and Chengnian Sun. 2023. On the Caching Schemes to Speed Up Program Reduction. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (January 2023), Article 1, 30 pages. <https://doi.org/doi/10.1145/3617172>
- [118] Yongqiang Tian, Xueyan Zhang, Yiwen Dong, Zhenyang Xu, Mengxiao Zhang, Yu Jiang, Shing-Chi Cheung, and Chengnian Sun. 2023. On the caching schemes to speed up program reduction. *ACM Transactions on Software Engineering and Methodology* 33, 1 (2023), 1–30. <https://doi.org/doi/10.1145/3617172>
- [119] Guancheng Wang, Ruobing Shen, Junjie Chen, Yingfei Xiong, and Lu Zhang. 2021. Probabilistic Delta debugging. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 881–892. <https://doi.org/10.1145/3468264.3468625>
- [120] Guancheng Wang, Ruobing Shen, Junjie Chen, Yingfei Xiong, and Lu Zhang. 2021. Probabilistic delta debugging. In *Proceedings of the 29th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 881–892. <https://doi.org/doi/10.1145/3468264.3468625>
- [121] Guancheng Wang, Ruobing Shen, Junjie Chen, Yingfei Xiong, and Lu Zhang. 2021. *ProbDD*. <https://github.com/Amocy-Wang/ProbDD> Accessed: 2023-09-28.
- [122] Haoyu Wang, Junjie Chen, Chuyue Xie, Shuang Liu, Zan Wang, Qingchao Shen, and Yingquan Zhao. 2023. Mlirsmith: Random program generation for fuzzing mlir compiler infrastructure. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1555–1566. <https://doi.org/doi/abs/10.1109/ASE56229.2023.00120>
- [123] Haijun Wang, Yun Lin, Zijiang Yang, Jun Sun, Yang Liu, Jinsong Dong, Qinghua Zheng, and Ting Liu. 2019. Explaining regressions via alignment slicing and mending. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2421–2437. <https://doi.org/10.1109/TSE.2019.2949568>
- [124] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superior: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software*

- Engineering (ICSE)*. IEEE, 724–735. <https://doi.org/10.1109/ICSE.2019.00081>
- [125] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering* (2024).
 - [126] Junjie Wang, Zhiyi Zhang, Shuang Liu, Xiaoning Du, and Junjie Chen. 2023. FuzzJIT: Oracle-Enhanced Fuzzing for JavaScript Engine JIT Compiler. In *USENIX Security Symposium*. USENIX. <https://doi.org/doi/abs/10.5555/3620237.3620342>
 - [127] Theodore Luo Wang, Yongqiang Tian, Yiwen Dong, Zhenyang Xu, and Chengnian Sun. 2023. Compilation Consistency Modulo Debug Information. In *ASPLOS '23: 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Vancouver, 25 March 2023 - 29 March 2023*. ACM. <https://doi.org/doi/10.1145/3575693.3575740>
 - [128] WebKit. 2001. *WebKit: Test Case Reduction*. <https://webkit.org/test-case-reduction/> Accessed: 2023-04-30.
 - [129] Dasarath Weeratunge, Xiangyu Zhang, William N Sumner, and Suresh Jagannathan. 2010. Analyzing concurrency bugs using dual slicing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*. 253–264. <https://doi.org/10.1145/1831708.1831740>
 - [130] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 172–184. <https://doi.org/10.1145/3611643.3616271>
 - [131] Wikipedia. 2022. *Jaccard index*. https://en.wikipedia.org/wiki/Jaccard_index Accessed: 2023-04-30.
 - [132] Wikipedia. 2022. *Standard Portable Intermediate Representation*. Retrieved 2023-01-25 from https://en.wikipedia.org/wiki/Standard_Portable_Intermediate_Representation

- [133] Wikipedia. 2022. *Static single-assignment form*. Retrieved 2022-08-31 from https://en.wikipedia.org/wiki/Static_single-assignment_form
- [134] Robert F Woolson. 2005. Wilcoxon signed-rank test. *Encyclopedia of Biostatistics* 8 (2005).
- [135] Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. 2023. How effective are neural networks for fixing security vulnerabilities. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1282–1294. <https://doi.org/doi/10.1145/3597926.3598135>
- [136] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. 187–204. <https://doi.org/doi/10.1145/2509578.2509581>
- [137] Chunqiu Steven Xia, Yifeng Ding, and Lingming Zhang. 2023. Revisiting the Plastic Surgery Hypothesis via Large Language Models. *arXiv preprint arXiv:2303.10494* (2023). <https://doi.org/10.48550/ARXIV.2303.10494>
- [138] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023)*. Association for Computing Machinery. <https://doi.org/10.1109/ICSE48619.2023.00129>
- [139] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for \$0.42 each using ChatGPT. *arXiv preprint arXiv:2304.00385* (2023). <https://doi.org/10.48550/arXiv.2304.00385>
- [140] Qi Xin, Myeongsoo Kim, Qirun Zhang, and Alessandro Orso. 2020. Program debloating via stochastic optimization. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*. 65–68. <https://doi.org/doi/10.1145/3377816.3381739>
- [141] Zhenyang Xu, Yongqiang Tian, Mengxiao Zhang, Jiarui Zhang, Puzhuo Liu, Yu Jiang, and Chengnian Sun. 2025. T-Rec: Fine-Grained Language-Agnostic Program Reduction Guided by Lexical Syntax. *ACM Transactions on Software Engineering and Methodology* 34, 2 (2025), 1–31. <https://doi.org/doi/10.1145/3690631>

- [142] Zhenyang Xu, Yongqiang Tian, Mengxiao Zhang, Gaosen Zhao, Yu Jiang, and Chengnian Sun. 2023. Pushing the Limit of 1-Minimality of Language-Agnostic Program Reduction. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 97 (apr 2023), 29 pages. <https://doi.org/10.1145/3586049>
- [143] Zhenyang Xu, Yongqiang Tian, Mengxiao Zhang, Gaosen Zhao, Yu Jiang, and Chengnian Sun. 2023. Pushing the Limit of 1-Minimality of Language-Agnostic Program Reduction. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 636–664. <https://doi.org/10.1145/3586049>
- [144] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 283–294. <https://doi.org/10.1145/1993316.1993532>
- [145] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200. <https://doi.org/10.1109/32.988498>
- [146] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. 2011. Localizing failure-inducing program edits based on spectrum information. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 23–32. <https://doi.org/document/6080769>
- [147] Mengxiao Zhang, Yongqiang Tian, Zhenyang Xu, Yiwen Dong, Shin Hwei Tan, and Chengnian Sun. 2024. Artifact for "LPR: Large Language Models-Aided Program Reduction". <https://doi.org/10.5281/zenodo.12669964>
- [148] Mengxiao Zhang, Yongqiang Tian, Zhenyang Xu, Yiwen Dong, Shin Hwei Tan, and Chengnian Sun. 2024. LPR: Large Language Models-Aided Program Reduction. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria, September 16–20). ACM, New York, NY, USA, 13. <https://doi.org/10.1145/3650212.3652126>
- [149] Mengxiao Zhang, Zhenyang Xu, Yongqiang Tian, Xinru Cheng, and Chengnian Sun. 2024. Artifact for "Toward a Better Understanding of Probabilistic Delta Debugging". <https://doi.org/10.5281/zenodo.14425529>
- [150] Mengxiao Zhang, Zhenyang Xu, Yongqiang Tian, Xinru Cheng, and Chengnian Sun. 2025. Toward a Better Understanding of Probabilistic Delta Debugging. arXiv:2408.04735 [cs.SE] <https://arxiv.org/abs/2408.04735>

- [151] Mengxiao Zhang, Zhenyang Xu, Yongqiang Tian, Yu Jiang, and Chengnian Sun. 2023. Artifact for "PPR: Pairwise Program Reduction". <https://doi.org/10.5281/zenodo.8267114>
- [152] Mengxiao Zhang, Zhenyang Xu, Yongqiang Tian, Yu Jiang, and Chengnian Sun. 2023. Artifact for "PPR: Pairwise Program Reduction". <https://doi.org/10.5281/zenodo.8267114>
- [153] Mengxiao Zhang, Zhenyang Xu, Yongqiang Tian, Yu Jiang, and Chengnian Sun. 2023. PPR: Pairwise Program Reduction. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023 (ESEC/FSE 2023)*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, New York, NY, USA. <https://doi.org/doi/10.1145/3611643.3616275>
- [154] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation*. 347–361. <https://doi.org/doi/10.1145/3140587.3062379>
- [155] Sai Zhang, Yu Lin, Zhongxian Gu, and Jianjun Zhao. 2008. Effective identification of failure-inducing changes: a hybrid approach. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. 77–83. <https://doi.org/10.1145/1512475.1512492>
- [156] Li Zhong and Zilong Wang. 2023. A study on robustness and reliability of large language model code generation. *arXiv preprint arXiv:2308.10335* (2023). <https://doi.org/10.48550/arXiv.2308.10335>
- [157] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Wenhai Li, Chao Ji, and Dan Ding. 2018. Delta debugging microservice systems. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*. 802–807. <https://doi.org/doi/10.1145/3238147.3240730>
- [158] Xintong Zhou, Zhenyang Xu, Mengxiao Zhang, Yongqiang Tian, and Chengnian Sun. 2025. WDD: Weighted Delta Debugging. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*.
- [159] Zhide Zhou, Zhilei Ren, Guojun Gao, and He Jiang. 2021. An empirical study of optimization bugs in GCC and LLVM. *Journal of Systems and Software* 174 (2021), 110884. <https://doi.org/10.1016/j.jss.2020.110884>