

Pushing the Limit of Language-Agnostic Program Reduction

by

Zhenyang Xu

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

© Zhenyang Xu 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: Ali Mesbah
 Professor, Department of Electrical and Computer Engineering
 The University of British Columbia

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

Internal Member: Joanne Atlee
 Professor, David R. Cheriton School of Computer Science
 University of Waterloo

Internal Member: Ondřej Lhoták
 Professor, David R. Cheriton School of Computer Science
 University of Waterloo

Internal-External Member: Arie Gurfinkel
 Professor, Department of 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

This thesis contains the following three published studies:

- **Pushing the Limit of 1-Minimality of Language-Agnostic Program Reduction**
Zhenyang Xu, Yongqiang Tian, Mengxiao Zhang, Gaosen Zhao, Yu Jiang, Chengnian Sun
Proceedings of the ACM on Programming Languages 7.OOPSLA1 (2023): 636-664
- **T-Rec: Fine-Grained Language-Agnostic Program Reduction Guided by Lexical Syntax**
Zhenyang Xu, Yongqiang Tian, Mengxiao Zhang, Jiarui Zhang, Puzhuo Liu, Yu Jiang, Chengnian Sun
ACM Transactions on Software Engineering and Methodology 34.2 (2025): 1-31
- **Boosting Program Reduction with the Missing Piece of Syntax-Guided Transformations**
Zhenyang Xu, Yongqiang Tian, Mengxiao Zhang, Chengnian Sun
Proceedings of the ACM on Programming Languages 9.OOPSLA2 (2025): 86-112

As the first author, I led all three studies and made major contributions to each of them. My contributions included (1) developing the research ideas, (2) designing and implementing the proposed approaches, (3) conducting experiments and analyzing the results, (4) writing the manuscripts and responding to comments from reviewers, and (5) presenting the work in the conference. My co-authors contributed by helping to refine the research ideas, offering suggestions on the experiment designs, and improving the manuscripts.

Abstract

Program reduction is a widely used technique for testing and debugging language processors. Given a program that triggers a bug in a language processor, program reduction searches for a canonical and minimal program that triggers the same bug, thereby facilitating bug deduplication and simplifying debugging. Among various reduction approaches, language-agnostic reducers (AGRs) have emerged as an important class of techniques because they do not rely on language-specific knowledge and can thus be applied across a wide range of programming languages. This generality makes AGRs especially valuable for languages lacking specialized reduction tools. However, previous AGRs support only a limited set of program transformations, which restricts their minimization and canonicalization capability and results in substantial performance gap compared to language-specific reducers (SPRs).

This thesis aims to enhance both the canonicalization and minimization capabilities of AGRs, thereby narrowing the performance gap between AGRs and SPRs. It comprises the following three contributions.

The first work aims to improve the reduction capability of AGRs by enabling them to integrate more transformations in an efficient way. As previously mentioned, previous AGRs support only a limited set of transformations. Once a 1-minimal result is obtained and no further transformation can reduce the program, the reduction process terminates. However, such a 1-minimal result may still contain excessive bug-irrelevant program elements. To address this limitation, this work proposes a framework named Vulcan. Vulcan employs an AGR as the main reducer and introduces a set of auxiliary reducers that perform diverse program transformations. When the main reducer can no longer make progress, Vulcan invokes one of its auxiliary reducers to create new reduction opportunities, and then re-applies the main reducer to further minimize the program. In addition to the framework, this work also presents three example program transformations: Identifier Replacement, Subtree Replacement, and Tree-Based Local Exhaustive Enumeration. Evaluation on a multilingual benchmark suite (referred to as **Benchmark-Reduce**) which includes C, Rust, and SMT-LIBv2 programs, demonstrates that Vulcan outperforms the state-of-the-art AGR, Perses, in terms of minimization. On average, Vulcan produces results with 33.55%, 21.61%, and 31.34% fewer tokens than Perses on C, Rust, and SMT-LIBv2 benchmarks, respectively.

The second work focuses on enhancing the canonicalization capability of AGRs. A reducer with strong canonicalization capability can minimize differences among programs that trigger the same bug, thereby greatly facilitating bug deduplication. However, prior

AGRs exhibit poor canonicalization capability, primarily because they treat tokens as atomic and irreducible units. To address this limitation, this work proposes T-Rec, a fine-grained, lexical syntax-guided program reduction technique that can effectively reduce and canonicalize each token in a program. Evaluation results show that integrating T-Rec into Vulcan enables the elimination of 1,315 additional duplicates in a benchmark suite containing 3,796 programs that expose 46 unique bugs (referred to as **Benchmark-Cano**). Moreover, T-Rec further reduces the size of Vulcan’s results on **Benchmark-Reduce** by up to 53.73% in terms of bytes.

The third work aims to further enhance both the minimization and canonicalization performance of AGRs by introducing additional program transformations. Specifically, this work proposes SFC, a novel syntax-guided transformation technique that has been overlooked by prior syntax-guided AGRs. To apply SFC effectively and efficiently in program reduction, three SFC-based reduction methods are designed: Smaller Structure Replacement, Identifier Elimination, and Structure Canonicalization. Evaluation results show that integrating these SFC-based methods into Vulcan yields an average 8.2% reduction in output size on **Benchmark-Reduce**. Moreover, when combined with T-Rec, the SFC-based methods enable Vulcan to eliminate an additional 435 duplicates in **Benchmark-Cano**.

Collectively, these studies significantly advance the effectiveness of language-agnostic program reduction in both minimization and canonicalization. By integrating the proposed approaches, the prior state-of-the-art AGR, Perses, can produce results that are on average 43% smaller on **Benchmark-Reduce** and eliminate 1,750 additional duplicates in **Benchmark-Cano**.

Acknowledgements

I would like to express my deepest gratitude to all the people who have guided me, collaborated with me, supported me, and accompanied me throughout my Ph.D. journey.

First and foremost, I am profoundly grateful to my supervisor, Prof. Chengnian Sun. He taught me the fundamental and essential skills required for conducting research in software engineering, gave me the freedom to explore topics that genuinely interested me, and consistently provided valuable feedback and suggestions. His mentorship has been invaluable to my growth not only as a researcher but also as an individual.

I would also like to sincerely thank my thesis committee members—Prof. Ali Mesbah, Prof. Joanne Atlee, Prof. Ondřej Lhoták, and Prof. Arie Gurfinkel—for their time and efforts. Additionally, I thank Prof. Yizhou Zhang, Prof. Meng Xu, Prof. Pengyu Nie and Prof. Shane McIntosh for serving on my comprehensive examination committee or attending my Ph.D. seminars.

My heartfelt appreciation goes to Prof. Yun Lin, who mentored me when I first began exploring research as an undergraduate student. He sparked my interest in research and connected me with Prof. Sun. Without his guidance, I would not have begun this journey.

I am grateful to Prof. Yu Jiang and his research group at Tsinghua University for welcoming me as a visiting student during the COVID-19 pandemic. I learned so much from the brilliant minds in the group and cherish the unforgettable experiences from that time.

My special thanks go to Prof. Yongqiang Tian and Dr. Yiwen Dong for their generous support and guidance throughout my Ph.D. journey. Yongqiang, who mentored me when he was a senior Ph.D. student in our group, has been both a "second supervisor" and a close friend. Yiwen, the first Ph.D. student in our group, offered tremendous help in both academic and personal aspects of life. Their mentorship and kindness made my Ph.D. experience much smoother.

I would like to thank all the friends I met in SWAG: Wenhan, Zhili, Xueyao, Mengxiao, Xueyan, Yaxin, Gengyi, Xintong, Hongxu, Chunhao, Yiran, Gaosen, Guoshuai, Noble, Vikram, Raymond, Boren, Farshad, Bihui, Mahtab, Aryan, Daniel, Amelia, Yelizaveta, Nimmi, Mahmoud, Mingyang, Xiaoyan, and many others. I apologize if I have missed anyone's name.

Additionally, I would like to thank my undergraduate roommates—Jingyu Pan, Shitao Fang, Wenzheng Ye, and Xiaotian Tang—for always being there to talk with. I also thank Runyu Wang, my roommate in Waterloo, for being my workout partner. Moreover, I

am deeply grateful to my girlfriend, Liner Ge, whose companionship and encouragement gave me the strength to overcome challenges. Meanwhile, her hard work and achievement continuously motivates me to improve myself.

Finally, I want to thank my family. My parents, Lihua Xu and Jinxue Xu, and my grandmother, Qinxian Lou, whose unconditional support and love have given me the courage to pursue my dreams. I also want to remember my grandfather, Liren Xu, who sadly could not see me complete my Ph.D. journey. His kindness and optimism continue to inspire me, and I carry his memory with me in every achievement.

Table of Contents

Examining Committee Membership	ii
Author’s Declaration	iii
Statement of Contributions	iv
Abstract	v
Acknowledgements	vii
List of Figures	xiii
List of Tables	xv
1 Introduction	1
1.1 Overview	2
1.2 Contributions	4
1.3 Organization	5
2 Background	6
2.1 Program Reduction	6
2.1.1 Canonicalization	6
2.1.2 Property Checker	7

2.1.3	Typical Workflow	8
2.1.4	Prior Language-Agnostic Reduction Techniques	8
3	Pushing the Limit of 1-Minimality of Language-Agnostic Program Reduction	10
3.1	Introduction	10
3.2	A Motivating Example	13
3.3	Background	17
3.3.1	Relaxed Goal in Program Reduction	17
3.4	Approach	18
3.4.1	Identifier Replacement	21
3.4.2	Sub-Tree Replacement	23
3.4.3	Tree-Based Local Exhaustive Enumeration	25
3.4.4	Order of Auxiliary Reducers	27
3.4.5	Parameters and Fixpoint Mode	27
3.5	Evaluation	29
3.5.1	RQ1: Further Reduction on 1-minimal Results	30
3.5.2	RQ2: Further Reduction on Language-Specific Tools' Results	31
3.5.3	RQ3: Generality of <i>Vulcan</i>	32
3.5.4	RQ4: Effectiveness of Each Auxiliary Reducer	35
3.5.5	RQ5: Performance with a More Aggressive Reduction Setting	36
3.6	Discussion	36
3.6.1	Impact of the Order of the Auxiliary Reducers	38
3.6.2	Impact of the Parameters	38
3.6.3	Threats to Validity	41
3.7	Related Work	41
3.8	Chapter Conclusion	43

4	T-Rec: Fine-Grained Language-Agnostic Program Reduction Guided by Lexical Syntax	51
4.1	Introduction	51
4.2	Motivating Example	54
4.3	Preliminaries	57
4.3.1	Canonicalization in C-Reduce	57
4.3.2	Lexical Syntax	57
4.4	Methodology	59
4.4.1	Definition of Canonical Tokens	60
4.4.2	Replacement-Based Token Canonicalization	62
4.4.3	Deletion-Based Token Canonicalization	65
4.5	Evaluation	70
4.5.1	RQ1: Effectiveness in Deduplication	74
4.5.2	RQ2: Effectiveness in Reducing Program Size in Bytes	76
4.5.3	RQ3: Effectiveness in Removing Tokens	79
4.5.4	RQ4: Efficiency	81
4.6	Discussion	84
4.6.1	Comparing to Perses with Lex Tree Extension	84
4.6.2	Threats to Validity	85
4.7	Related Work	86
4.7.1	Language-Agnostic Program Reducers	86
4.7.2	Language-Specific Program Reducers	88
4.8	Chapter Conclusion	89
5	Boosting Program Reduction with the Missing Piece of Syntax-Guided Transformations	90
5.1	Introduction	90
5.2	Preliminary	93

5.2.1	Syntax of Programming Languages	94
5.2.2	Syntax-Guided Transformation	96
5.3	Methodology	97
5.3.1	Structure Form Conversion (SFC)	97
5.3.2	Program Reduction with SFC	102
5.3.3	Implementation	106
5.4	Evaluation	107
5.4.1	RQ1: Performance in Minimization	110
5.4.2	RQ2: Performance in Canonicalization	112
5.4.3	RQ3: Ablation Studies	115
5.5	Related Work	119
5.6	Chapter Conclusion	121
6	Conclusion and Future Work	122
6.1	Conclusion	122
6.2	Future Work	123
	References	124

List of Figures

2.1	A typical workflow of a program reducer.	7
3.1	A set of programs that trigger the crash bug LLVM-26760. (a) is the 1-tree-minimal result reduced by Perses from the original bug-triggering program. (b) is obtained by replacing a single identifier of (a) and applying Perses on the modified program again. (c) is obtained from (b) by changing a type specifier to a different one and applying Perses again. (d) is obtained from (c) by removing two nodes in its tree representation simultaneously. (e) is the result produced by Vulcan	15
3.2	The workflow of Vulcan	18
3.3	An illustrative example that shows how the sliding window (marked in orange box, length = 3) moves during the process of tree-based local exhaustive search. If the window has not reached the end of the current node sequence (marked with red), it moves forward by one node (step ① and ③). Otherwise, the node sequence is updated by replacing all non-leaf nodes with their children (step ②).	26
3.4	Evaluation results of Perses and Vulcan on SMT-LIBv2 benchmarks. . . .	34
3.5	Evaluation results of Vulcan using different orders of auxiliary reducers on the C benchmarks.	37
3.6	Evaluation results of Vulcan with different σ_{limit} on the C benchmarks. . .	39
3.7	Evaluation results of Vulcan with different σ_{wsize} on the C benchmarks. . .	40
4.1	Code snippets in programs output by Perses, C-Reduce-cano _{Perses} , and T-Rec _{Perses} on benchmark rust-77002 (left) and gcc-60116 (right).	55
4.2	Results output by Perses and T-Rec _{Perses} on benchmark gcc-71626.	56

4.3	The partial lexical grammar of Constant token in C written in ANTLR grammar format	58
4.4	The overview of T-Rec.	59
4.5	An example of property checking with a new token	65
4.6	The lex tree of token 0xff00ull in C language.	65
4.7	An example showing the process of the deletion-based token canonicalization.	69
4.8	The program sizes in bytes of the results produced by different reducers on SMT-LIBv2 benchmarks. On average, the results of T-Rec _{Perses} are 42.86% and 35.39% smaller than Perses and C-Reduce-cano _{Perses} , respectively; the results of T-Rec _{Vulcan} are 16.24% and 4.83% smaller than Vulcan and C-Reduce-cano _{Vulcan} , respectively.	78
4.9	The token counts of the results produced by different reducers on SMT-LIBv2 benchmarks. On average, the results of T-Rec _{Perses} contain 34.35% fewer tokens than Perses.	81
4.10	The execution time of different reducers on SMT-LIBv2 benchmarks. On average, T-Rec _{Perses} takes 56.46% and 22.04% more time than Perses and C-Reduce-cano _{Perses} , respectively; T-Rec _{Vulcan} takes 5.01% more time than Vulcan.	83
5.1	An example CFG and an example parse tree.	94
5.2	The overall workflow of SFC _{Perses}	107
5.3	Distribution of the original program sizes in Benchmark-Reduce	108
5.4	Evaluation results of Perses, Vulcan, SFC _{Perses} and SFC _{Vulcan} on C benchmarks.	115
5.5	Evaluation results of Perses, Vulcan, SFC _{Perses} and SFC _{Vulcan} on Rust benchmarks.	117
5.6	Evaluation results of Perses, Vulcan, SFC _{Perses} and SFC _{Vulcan} on SMT-LIBv2 benchmarks.	118

List of Tables

3.1	Number of queries required by performing Tree-Based Local Exhaustive Enumeration with different values of σ_{wsize} . $Q_{pos}(\sigma_{wsize})$ is the number of queries required by each position of the sliding window given the σ_{wsize} . $Q_{level}(\sigma_{wsize})$ is the number of queries required by the lowest level of a parse tree that has s leaf nodes given the σ_{wsize}	45
3.2	Evaluation results of Perses, C-Reduce, and Vulcan on C programs. The size of a program is measure by the number of tokens it contains. $O(\#)$ is the size of the O riginal bug-triggering program. $R(\#)$ is the size of the R educed program. $Q(\#)$ is the number of required property test Q ueries. $T(s)$ is the execution T ime in seconds. $C(\%)$ is the percentage C hanges in size of V ulcan's results <i>w.r.t.</i> Perses and C-Reduce.	46
3.3	Evaluation results of Vulcan and Perses on C-Reduce reduced C programs. $C(\%)$ is the percentage C hange in size <i>w.r.t.</i> the result of C-Reduce.	47
3.4	Evaluation results of Perses, C-Reduce, and Vulcan on Rust programs. $C(\%)$ is the percentage C hanges in size of V ulcan's results <i>w.r.t.</i> Perses and C-Reduce.	48
3.5	Results of Vulcan _{EE} , Vulcan _{IR} and Vulcan _{SR} reducing the 1-tree-minimal program output by Perses. $E(\#)$ is the number of E liminated tokens in reduction.	49
3.6	Evaluation results of Vulcan and Vulcan ⁺ on C program benchmarks. $C(\%)$ is the size change percentages of Vulcan ⁺	50
4.1	The reduction results produced by different reducers on the 2,501 crash test cases in Benchmark-Cano . The column Eliminated ($\#$) shows the total number of programs eliminated by each reducer.	73

4.2	The reduction results produced by different reducers on the 1,295 miscompilation test cases in Benchmark-Cano . The Eliminated (#) column shows the total number of programs that are eliminated because they are textually equivalent to other programs.	74
4.3	The program sizes in bytes of the results produced by different reducers on C and Rust benchmarks. CR represents C-Reduce-cano.	77
4.4	The token counts of the results produced by different reducers on C and Rust. CR represents C-Reduce-cano.	80
4.5	The execution time of different reducers on C and Rust. CR represents C-Reduce-cano.	82
4.6	The reduction results produced by different reducers on the 2,501 crash test cases in Benchmark-Cano . The column Eliminated (#) shows the total number of programs eliminated by each reducer.	85
5.1	Evaluation results of Perses, Vulcan , and SFC_{Perses} on C and Rust benchmarks. Column $\% \Delta$ shows the average percentage change in size achieved compared to the baseline reducer.	113
5.2	Evaluation results of C-Reduce, Vulcan , and SFC_{Vulcan} on C and Rust benchmarks. Column $\% \Delta$ shows the average percentage change in size achieved compared to the baseline reducer.	114
5.3	Evaluation results of Perses, SFC_{Perses} , SFC_{Perses}^{--sc} , Vulcan , SFC_{Vulcan} , and SFC_{Vulcan}^{--sc} on Benchmark-Cano . Each reducer is integrated with T-Rec. The column Eliminated shows the total number of eliminated duplicates. The column Size and Time shows the average size of the outputs and the average execution time.	116
5.4	Results of SFC_{Perses}^{--ie} , SFC_{Perses}^{--ssr} , SFC_{Vulcan}^{--ie} , and SFC_{Vulcan}^{--ssr} on Benchmark-Reduce . Column Worse is the number of cases where the removal causes a deterioration in minimization capability. Column $\% \Delta$ is the average percentage increase in size caused by the removal.	116

Chapter 1

Introduction

Language processors such as compilers and interpreters play a vital role in software development. Their correctness and robustness are essential for efficiently building reliable software. However, as one of the most complex types of software systems, language processors are highly prone to bugs. Extensive research studies have been conducted to systematically and automatically test language processors and to identify bugs effectively and efficiently [117, 223, 139, 136, 13, 67, 219, 189, 231, 46, 218]. These studies have made significant contributions by detecting large numbers of bugs across various language processors such as GCC, LLVM, Rustc, and SMT-LIBv2 solvers.

While the increasing number of bug reports helps maintainers and developers steadily improve language processors, it also exposes and amplifies the challenges inherent in the debugging process. Specifically, bug-triggering input programs generated by existing approaches, *e.g.*, fuzzing, often contain substantial amounts of bug-irrelevant code. This leads to two major issues. First, bug-irrelevant code distracts maintainers and increases the difficulties to pinpoint the root cause of a bug. Second, during automated testing, different generated programs may essentially trigger the same underlying bug, necessitating bug deduplication. Excessive bug-irrelevant code can significantly deteriorate the performance of existing bug deduplication techniques [45, 222]. Therefore, a technique that can automatically remove bug-irrelevant code from bug-triggering programs is extremely valuable and highly desired. Many compilers and interpreters, such as GCC, Clang, CPython and JerryScript, have explicitly required bug reporters to reduce the bug-triggering programs before submitting bug reports [74, 140, 103, 154, 52]. This motivates the studies on program reduction. A prior study shows that 80% of bugs in GCC and LLVM can be revealed by a program containing fewer than 45 lines of code [191], demonstrating the strong potential of program reduction.

In recent years, many program reduction techniques have been proposed [226, 152, 192, 161, 114, 170, 109, 110]. Some of these proposed reducers are designed without incorporating any language-specific knowledge [226, 152, 192]. Such reducers are classified as language-agnostic program reducers (AGRs). They are applicable to a wide range of programming languages and can serve as a starting point for building a more powerful language-specific reducer (SPRs), making them particularly valuable for languages lacking specialized reduction tools.

However, prior AGRs still exhibit significant limitations. First, although state-of-the-art AGRs can efficiently reduce extremely large programs (containing hundreds of thousands of tokens) to much smaller ones (containing hundreds or even tens of tokens), sometimes their results may still contain excessive bug-irrelevant code. There is no effective approach for further reducing these reduced results. Second, prior AGRs treat tokens as atomic elements and overlook reduction opportunities within individual tokens. This restricts their canonicalization capability, which is important for enabling effective bug deduplication. Third, the transformations supported by prior AGRs are limited, constraining their overall reduction capability.

1.1 Overview

This thesis includes three studies to enhance language-agnostic program reduction. Specifically, the thesis statement is:

Thesis Statement. This thesis introduces a novel language-agnostic program reduction framework and a diverse suite of reduction transformations that jointly advance minimization and canonicalization. Together, integrating these techniques to prior state of the art achieves substantially smaller and more canonical reduced programs.

This thesis comprises three studies that progressively contribute to this statement. The first study introduces a novel language-agnostic program reduction framework named *Vulcan*, which enables the integration of diverse transformations to achieve further reduction while keeping computational overhead manageable. The second study introduces a fine-grained reduction technique to reduce and canonicalize individual tokens in the program, thereby enhancing reduction capability of AGRs, particularly in terms of canonicalization. The last study proposes a novel syntax-guided transformation named SFC along with three reduction methods based on SFC, which further improve both minimization and canonicalization capability of AGRs.

A Framework for Efficiently Enabling Further Reduction with Diverse Trans-

formations. The first work aims to improve the reduction capability of AGRs by enabling them to integrate more transformations in an efficient way. Previous AGRs support only a limited set of transformations. Once a 1-minimal result is obtained and no further transformation can reduce the program, the reduction process terminates. However, such a 1-minimal result may still contain excessive bug-irrelevant program elements. To address this limitation, this work proposes a framework named Vulcan. Vulcan employs an AGR as the main reducer and introduces a set of auxiliary reducers that perform diverse program transformations. When the main reducer can no longer make progress, Vulcan invokes one of its auxiliary reducers to create new reduction opportunities, and then re-applies the main reducer to further minimize the program. In addition to the framework, this work also presents three example program transformations: Identifier Replacement, Subtree Replacement, and Tree-Based Local Exhaustive Enumeration. Evaluation on a multilingual benchmark suite (referred to as **Benchmark-Reduce**) which includes C, Rust, and SMT-LIBv2 programs, demonstrates that Vulcan outperforms the state-of-the-art AGR, Perses, in minimization capability. On average, Vulcan produces results with 33.55%, 21.61%, and 31.34% fewer tokens than Perses on C, Rust, and SMT-LIBv2 benchmarks, respectively. This study was published in Proceedings of the ACM on Programming Languages (OOPSLA 2023).

- **Pushing the Limit of 1-Minimality of Language-Agnostic Program Reduction**

Zhenyang Xu, Yongqiang Tian, Mengxiao Zhang, Gaosen Zhao, Yu Jiang, Chengnian Sun

Proceedings of the ACM on Programming Languages 7.OOPSLA1 (2023): 636-664

A Fine-Grained Lexical Syntax-Guided Reduction Technique for Token Reduction.

The second work focuses on enhancing the canonicalization capability of AGRs. A reducer with strong canonicalization capability can minimize differences among programs that trigger the same bug, thereby greatly facilitating bug deduplication. However, prior AGRs exhibit poor canonicalization capability, primarily because they treat tokens as atomic and irreducible units. To address this limitation, this work proposes T-Rec, a fine-grained, lexical syntax-guided program reduction technique that can effectively reduce and canonicalize each token in a program. Evaluation results show that integrating T-Rec into Vulcan enables the elimination of 1,315 additional duplicates in a benchmark suite containing 3,796 programs that expose 46 unique bugs (referred to as **Benchmark-Cano**). Moreover, T-Rec further reduces the size of Vulcan’s results on **Benchmark-Reduce** by up to 53.73% in terms of bytes. This study was published in ACM Transactions on Software Engineering and Methodology (TOSEM) in 2024.

- **T-Rec: Fine-Grained Language-Agnostic Program Reduction Guided by Lexical Syntax**

Zhenyang Xu, Yongqiang Tian, Mengxiao Zhang, Jiarui Zhang, Puzhuo Liu, Yu Jiang, Chengnian Sun

ACM Transactions on Software Engineering and Methodology 34.2 (2025): 1-31

A Novel Syntax-Guided Transformation for Effective Reduction. The third work aims to further enhance both the minimization and canonicalization capability of AGRs by introducing additional program transformations. Specifically, this work proposes SFC, a novel syntax-guided transformation technique that has been overlooked by prior syntax-guided AGRs. To apply SFC effectively and efficiently in program reduction, three SFC-based reduction methods are designed: Smaller Structure Replacement, Identifier Elimination, and Structure Canonicalization. Evaluation results show that integrating these SFC-based methods into Vulcan yields an average 8.2% reduction in output size on **Benchmark-Reduce**. Moreover, when combined with T-Rec, the SFC-based methods enable Vulcan to eliminate an additional 435 duplicates in **Benchmark-Cano**. This study was published in Proceedings of the ACM on Programming Languages (OOPSLA 2025).

- **Boosting Program Reduction with the Missing Piece of Syntax-Guided Transformations**

Zhenyang Xu, Yongqiang Tian, Mengxiao Zhang, Chengnian Sun

Proceedings of the ACM on Programming Languages 9.OOPSLA2 (2025): 86-112

1.2 Contributions

This thesis makes three main contributions to enhancing the capability of AGRs.

A Framework for Further Reduction. We propose Vulcan, a language-agnostic program reduction framework that efficiently enables further reduction after obtaining a 1-minimal result by applying additional transformations. Our evaluation demonstrates that Vulcan greatly outperforms the state-of-the-art tool in reduction capability. We have released the implementation and data associated with this study at <https://zenodo.org/records/8197652>

A Fine-Grained Reduction Technique. We propose T-Rec, a fine-grained language-agnostic program reduction technique which leverages the lexical syntax of programming languages to effectively exploit reduction opportunities within each token. Our evaluation

shows that T-Rec significantly enhances the reduction capability, particularly in terms of canonicalization. We have released the implementation and data associated with this study at <https://github.com/trec-reducer/T-Rec>

A Novel Syntax-Guided Transformation for Reduction. We propose a novel syntax-guided transformation named SFC, which guarantees syntactically valid outputs after transformation. Based on SFC, we design three reduction methods. Our evaluation reveals that these SFC-based reduction methods can further improve the reduction capability of AGRs. We have released the implementation and data associated with this study at <https://github.com/sfc-reducer/sfc-reducer>.

1.3 Organization

The remainder of the thesis is organized as follows. Chapter 2 introduces the background knowledge, including the definition and goals of program reduction and syntax of programming languages. Chapter 3 introduces *Vulcan*, the proposed reduction framework for enabling further reduction. Chapter 4 presents T-Rec, a fine-grained reduction approach for reducing each token internally. Chapter 5 describes SFC, a novel syntax-guided transformation and the reduction approaches designed based on it. Chapter 6 summarizes this thesis and discusses future work.

Chapter 2

Background

2.1 Program Reduction

Let $\mathbb{P}$ denote the search space of program reduction, namely, the set of all possible programs during reduction. Given a property checking function $\psi : \mathbb{P} \rightarrow \{\text{true}, \text{false}\}$ that can verify whether a program possesses a certain property, and a program that possesses the specific property, *i.e.*, $P \in \mathbb{P}$ *s.t.* $\psi(P)$, program reduction aims to output a minimized bug-triggering program P_{min} by keeping searching for another program $p \in \mathbb{P}$ *s.t.* $\psi(p) \wedge |p| < |P|$, ($|P|$ is the size of program P). In this thesis, we use the number of tokens and the byte size as the metrics for measuring program sizes. The two metrics are also adopted in prior closely related studies [152, 192, 170].

A primary application scenario of program reduction is minimizing bug-triggering programs. In this setting, the input program is a bug-triggering program, the input property checking function is a script that checks whether the program still triggers the bug under investigation, and the output is a minimized program that continues to reproduce the bug.

2.1.1 Canonicalization

In addition to minimizing bug-triggering programs, another important and highly desired feature of program reduction is canonicalization [87, 144]. Ideally, given a property ψ and a search space $\mathbb{P}$ that includes all the possible programs, a perfectly canonicalizing program

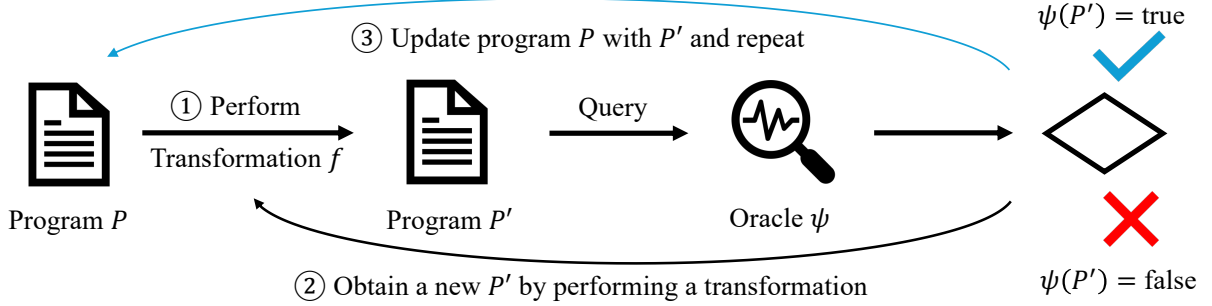


Figure 2.1: A typical workflow of a program reducer.

reduction algorithm f has the following property:

$$|\{f(P, \psi) | P \in \mathbb{P} \wedge \psi(P)\}| = 1 \quad (2.1)$$

However, achieving perfect canonicalization is impractical [87, 45]. Therefore, a more attainable goal is often pursued: given a set of duplicates, the program reducer should minimize the differences among them as much as possible and reduce them to a minimal set of unique programs (*i.e.*, minimizing $|\{f(P, \psi) | P \in \mathbb{P} \wedge \psi(P)\}|$). This canonicalization capability is highly valuable in practice. For example, during a large-scale automated testing campaign for a language processor, a significant amount of bug-triggering programs may be generated. In such a case, a highly canonicalizing reducer could eliminate most duplicates and help address the fuzzer taming problem [45], *i.e.*, rank bug-triggering programs so that those triggering distinct bugs appear early in the list. From a debugging perspective, canonicalization can reduce the need for developers to examine numerous duplicates that vary only in unimportant ways, thereby simplifying and accelerating the debugging process [87].

2.1.2 Property Checker

A property checker (or oracle) ψ , is crucial to the effectiveness of program reduction. The quality of the property checker has a significant impact on the reduction’s performance and results. In practice, a property checker is typically implemented as a script. To enable effective reduction, the checker must be sufficiently comprehensive. For example, instead of validating the program using only the buggy compiler, a common practice is to compile and run the program using multiple compilers or multiple versions of the same compiler, including both buggy and correct ones. In addition, for C/C++ programs, the checker

should verify not only the presence of the target bug symptom but also the absence of undefined behaviors.

In practice, to reduce engineering effort while maintaining high-quality property checking, a template-based approach is commonly adopted. For a specific language and bug type, a template property-checking script is first designed. When a new instance of the same bug type is discovered, a corresponding property checker can then be automatically generated by instantiating the template with bug-specific information. For example, in compiler testing tools such as Kitten [199, 219], when a new C compiler crash bug is discovered, the property checker is generated by filling in placeholders with the relevant compiler versions and key error signatures (*e.g.*, stack traces or diagnostic messages associated with the crash).

2.1.3 Typical Workflow

Program reduction is typically a trial-and-error process. As shown in Figure 2.1, a program reducer first performs a transformation $f_t : \mathbb{P} \rightarrow \mathbb{P}$ to the original program P and obtains a different program p (step ①). If p does not trigger the same bug as P does (*i.e.*, $\psi(p) = \text{false}$), p is discarded and the reducer will attempt to obtain another different program by performing another transformation and repeat the process (step ②). Otherwise, if $\psi(p) = \text{true}$, which means a smaller program is successfully found, the reduction continues by treating p as the original program P and searching for the next smaller program (step ③).

Fixpoint. During the reduction process, it is common that deleting some elements makes some other elements deletable. To achieve the best reduction result within the capacity of the transformations, a typical way is to repeatedly apply each transformation to its output until no progress can be made by any transformation [152, 192, 170, 97, 112, 207]. Such a program that cannot be minimized by the reducer is called a fixpoint of the program reducer.

2.1.4 Prior Language-Agnostic Reduction Techniques

Minimizing Delta Debugging. The minimizing delta debugging algorithm `ddmin` [226] is a fundamental algorithm used by a wide range of program reduction tools. Given a bug-triggering input, the intuition of `ddmin` is to consider the input as a list of elements and keep testing whether a partition of the list or the complement of this partition can still

trigger the same bug. `ddmin` starts by dividing the list into two partitions and gradually decreases the granularity when none of the partitions or complements can trigger the same bug.

Tree-Based Program Reduction. Tree-based program reduction leverages the tree structure of the program to enhance the reduction performance. The input program is first converted to its tree representation (*e.g.*, its parse tree), and the program reduction algorithm is applied to this tree structure instead of the plain text of the program. For example, HDD runs `ddmin` on each level of the tree representation in a top-down manner.

Syntax-Guided Program Reduction. Besides leveraging the tree structure of programs, syntax-guided program reduction also utilizes formal syntax to guide the reduction. Program reduction is a trial-and-error process. During this process, a large number of programs are generated by deleting part of the original program and tested against the property. The key insight of syntax-guided program reduction is that syntactically invalid programs usually do not have the property (*i.e.*, triggering the same bug as the original program). Therefore, by avoiding generating invalid programs and performing property tests on only syntactically valid ones, much time can be saved.

Chapter 3

Pushing the Limit of 1-Minimality of Language-Agnostic Program Reduction

This chapter introduces **Vulcan**, a language-agnostic program reduction framework that augments a main reducer with auxiliary reducers capable of performing diverse, strategically selected transformations. In addition, this chapter presents three example auxiliary reducers. The evaluation shows that, by alternately invoking the main reducer and these auxiliary reducers, **Vulcan** overcomes the stagnation commonly encountered after achieving a 1-minimal result. The framework effectively enables further reduction and demonstrates significant improvements over the state-of-the-art AGR, Perses, across C, Rust, and SMT-LIBv2 programs.

3.1 Introduction

The goal of program reduction is to find the minimal program that still exhibits the given property ψ . However, obtaining the global minimum is NP-complete even if the check for property ψ is constant-time [226, 152]. To make program reduction practical, all existing program reducers resort to producing 1-minimal results, a type of local minima first proposed by Zeller and Hildebrandt [226]. A reduced program is 1-minimal if and only if deleting any single element from the program makes the program lose the property ψ . The 1-minimality has been demonstrated to be an effective goal. For example, it enables the state-of-the-art AGR Perses to reduce programs from hundreds of thousands of tokens to

just hundreds or even tens of tokens quickly [192].

Problems with 1-minimality of AGRs. However, the 1-minimal results produced by AGRs can still be large and contain excessive bug-irrelevant program elements, making the reduced programs suboptimal for debugging. Therefore, it remains desirable but also challenging to further reduce results by AGRs beyond 1-minimality. One feasible solution is to design and implement SPRs. For example, C-Reduce tackles this problem by leveraging well-crafted domain-specific program transformations to reduce C/C++ programs [170]. These language-specific transformations, such as source-to-source function inlining, transform programs in different ways from how AGRs do. As a result, C-Reduce can output much smaller results than those from HDD or Perses when reducing C/C++ programs. However, constructing such a language-specific tool requires extensive language-specific knowledge and significant engineering efforts, and the crafted tool usually can work effectively for only one specific programming language. In fact, to the best of our knowledge, only C/C++, Java, and SMT-LIBv2 have C-Reduce, J-Reduce [110], and ddSMT [161, 114] as their language-specific reducers; other programming languages do not have such counterparts.

Vulcan. In this chapter, we propose *Vulcan*, a *language-agnostic* reduction framework to push the limit of 1-minimality of AGRs. *Vulcan* can further minimize the results of AGRs by exploiting the formal syntax of the language to perform diverse program transformations, in hope of creating reduction opportunities for other reduction algorithms to progress or even directly deleting bug-irrelevant elements from the results. The key insights of this work are two-fold.

1. The program transformations in all existing program reducers, especially AGRs, are not diverse enough. To the best knowledge of the authors, all previously proposed AGRs only support deletion-based program transformations, *i.e.*, program transformations that only delete elements from the original program. This limitation traps AGRs early in 1-tree-minimality, and is the main reason that AGRs do not perform as well as SPRs.
2. Compared to the original bug-triggering program which might contain hundreds of thousands of tokens, the 1-minimal result produced by AGRs are relatively small; thus, performing more deletion-based program transformations with different deletion strategies and even non-deletion-based program transformations *e.g.*, program transformations that do not necessarily reduce the sizes of the programs, becomes time-wise affordable.

Vulcan incorporates an AGR as the main reducer and a series of auxiliary reducers to keep pushing the limit of 1-minimality of the main reducer. Given a program to be reduced, *Vulcan* first invokes the main reducer to produce a 1-minimal result. Then, one of the

auxiliary reducers is invoked to perform program transformations in hope of directly deleting nodes from the 1-minimal result or creating reduction opportunities. Next, the new program produced by the auxiliary reducer is sent back to the main reducer for further reduction until 1-minimality is achieved again. In this way, **Vulcan** can effectively further minimize the results of AGRs with reasonable computational cost.

Accompanied with our framework, we proposed the following three language-agnostic program transformations as examples.

Identifier Replacement. A large number of unique identifiers may lead to complex data-flow and control-flow in a program, which is hard to analyze. By minimizing the number of unique identifiers, a bug-triggering program may be simplified and reduced to a smaller size, which can further facilitate the debugging of language processors. With this insight, we propose *Identifier Replacement* to further minimize the number of unique identifiers and the size of the program.

Sub-Tree Replacement. This program transformation is inspired by mutation-based compiler fuzzing, which is a testing technique that randomly generates test programs by mutating existing programs to test the target compiler. Replacing a sub-tree of the program with a new one is a typical mutation operation applied in many existing mutation-based compiler testing studies [210, 13]. During a fuzzing process, it is common that different generated test programs trigger the same bug. Therefore, we believe by replacing a sub-tree of a 1-minimal program, it is likely to derive different programs inducing the same bug, and such programs might either be smaller or bring potential reduction opportunities.

Tree-Based Local Exhaustive Enumeration. A straightforward approach to further reducing 1-minimal result is changing the reduction goal to n -minimality, where $n > 1$. Specifically, we can further reduce 1-minimal result by enumerating all the programs that can be derived by deleting m nodes, where $m \leq n$ and then performing property tests on them. However, the computational cost of such an approach is too expensive, thus being impractical. To overcome this obstacle, we propose *Tree-Based Local Exhaustive Enumeration*. This program transformation restricts the exhaustive enumeration in a fixed-length sliding window, which slides through each level of the tree representation of the program.

Our extensive evaluation shows **Vulcan** is effective while being general. First, the experiment conducted with 20 bug-triggering C programs demonstrated that **Vulcan** can not only effectively further reduce 1-tree-minimal result produced by the state-of-the-art AGR Perses but also further reduce results produced by C-Reduce, the state-of-the-art SPR in reducing C programs. On average, the output of **Vulcan** contains 33.55% fewer tokens than that of Perses on the C program benchmarks, and **Vulcan** can further reduce the

size of C-Reduce’s results on these benchmarks by 10.07% on average. Second, evaluation results on Rust programs and SMT-LIBv2 programs demonstrate the generality of **Vulcan** on diverse languages. On average, the output of **Vulcan** contains 21.61% and 31.34% fewer tokens than that of Perses on Rust benchmarks and SMT-LIBv2 benchmarks respectively. Moreover, we conducted ablation studies to comprehend the contribution of each proposed program transformation towards breaking the limit of 1-tree-minimality. The results show all the proposed transformations are helpful to further reduce the program after achieving 1-tree-minimality. At last, by comparing **Vulcan** with **Vulcan**⁺, a variant with more aggressive reduction settings than the default **Vulcan**, we demonstrated that **Vulcan** can further trade off execution time for a smaller result. On average, the output of **Vulcan**⁺ contains 5.89% fewer tokens than that of **Vulcan** at the cost of taking $1.81\times$ the execution time of **Vulcan** to finish.

Contribution. We make the following contributions.

- We propose a language-agnostic program reduction framework named **Vulcan**, which can further minimize the results of AGRs. It is the first language-agnostic program reduction technique that performs diverse program transformations by exploiting the formal syntax of the language to eventually help produce smaller reduction results.
- Accompanied with the framework, we propose three simple but effective examples of language-agnostic program transformations and analyze their computational complexity.
- We implemented a prototype of **Vulcan** on top of Perses, the state-of-the-art language-agnostic program reduction tool, with all the proposed program transformations incorporated. By conducting comprehensive experiments with multilingual benchmarks of C, Rust and SMT-LIBv2 programs, the effectiveness and generality of **Vulcan** are strongly demonstrated.

3.2 A Motivating Example

In this section, we use a compiler crash bug LLVM-26760 [41] as a motivating example to elaborate how **Vulcan** further reduces a 1-minimal program produced by Perses.

Figure 3.1a: Result of Perses. Figure 3.1a shows the program reduced by Perses from the original bug-triggering program for LLVM-26760. Despite being 1-minimal, the program still has 116 tokens remaining for compiler developers to analyze. If this program

```

1  typedef signed int8_t;
2  typedef short int16_t;
3  typedef int int32_t;
4  typedef unsigned uint32_t;
5  int8_t g_100;
6  int16_t func_33() {
7      int8_t l_790;
8      int32_t l_919 = 0x24F96B7BL;
9      uint32_t l_1052;
10     if (l_790)
11         for (;;)
12             break;
13     else for (; l_919; --l_919);
14     int32_t l_1081 = 1L;
15     int32_t B4o4obl_919 = l_919;
16     // The following three assignments
17     // and line 24 essentially pass
18     // the value of g_100 to l_790
19     int8_t B4o4ocg_100 = g_100;
20     int32_t B4o4odl_1369 = B4o4ocg_100;
21     uint32_t B4o4ofl_1433 = B4o4odl_1369;
22     LABEL_4o4og;;
23     l_1052 = l_1052 >> l_1081;
24     l_790 = B4o4ofl_1433;
25     // if we can replace the l_790 on line 28,
26     // we can delete all the assignments
27     // mentioned above
28     if (l_790) {
29         l_1052 = l_1052 << B4o4obl_919;
30         goto LABEL_4o4og;
31     }
32 }
33 int main() {}

```

(a) 1-tree-minimal result by Perses.

```

1  typedef signed int8_t;
2  typedef short int16_t;
3  typedef int int32_t;
4  typedef unsigned uint32_t;
5  int8_t g_100;
6  int16_t func_33() {
7      int8_t l_790;
8      int32_t l_919 = 0x24F96B7BL;
9      uint32_t l_1052;
10     if (l_790)
11         for (;;)
12             break;
13     else for (; l_919; --l_919);
14     int32_t l_1081 = 1L;
15     int32_t B4o4obl_919 = l_919;
16     LABEL_4o4og;;
17     l_1052 = l_1052 >> l_1081;
18     if (g_100) {
19         l_1052 = l_1052 << B4o4obl_919;
20         goto LABEL_4o4og;
21     }
22 }
23 int main() {}

```

(b) Result obtained from (a) by replacing l_790 in orange with g_100 and re-applying Perses.

```

1  typedef signed int8_t;
2  typedef int int32_t;
3  typedef unsigned uint32_t;
4  int8_t g_100;
5  short func_33() {
6      int8_t l_790;
7      int32_t l_919 = 0x24F96B7BL;
8      uint32_t l_1052;
9      if (l_790) for (;;) break;
10     else for (; l_919; --l_919);
11     int32_t l_1081 = 1L;
12     int32_t B4o4obl_919 = l_919;
13     LABEL_4o4og;;
14     l_1052 = l_1052 >> l_1081;
15     if (g_100) {
16         l_1052 = l_1052 << B4o4obl_919;
17         goto LABEL_4o4og;
18     }
19 }
20 int main() {}

```

(c) Result obtained from (b) by replacing int16_t in orange with short and re-applying Perses.

```

1  typedef signed int8_t;
2  typedef int int32_t;
3  typedef unsigned uint32_t;
4  int8_t g_100;
5  short func_33() {
6      int8_t l_790;
7      int32_t l_919 = 0x24F96B7BL;
8      uint32_t l_1052;
9      if (l_790) for (; l_919; --l_919);
10     int32_t l_1081 = 1L;
11     int32_t B4o4obl_919 = l_919;
12     LABEL_4o4og;;
13     l_1052 = l_1052 >> l_1081;
14     if (g_100) {
15         l_1052 = l_1052 << B4o4obl_919;
16         goto LABEL_4o4og;
17     }
18 }
19 int main() {}

```

(d) Result obtained from (c) by deleting `for` statement and the keyword `else`.

```

1  typedef unsigned uint32_t;
2  short l_1052() {
3      uint32_t main = 0x24F96B7BL;
4      uint32_t l_1052;
5      for (; l_1052; --main)
6          ;
7      uint32_t uint32_t = 1L;
8      main;;
9      l_1052 = l_1052 >> uint32_t;
10     l_1052 = l_1052 << main;
11     goto main;
12 }
13 int main() {}

```

(e) Result produced by `Vulcan`.

Figure 3.1: A set of programs that trigger the crash bug LLVM-26760. (a) is the 1-tree-minimal result reduced by Perses from the original bug-triggering program. (b) is obtained by replacing a single identifier of (a) and applying Perses on the modified program again. (c) is obtained from (b) by changing a type specifier to a different one and applying Perses again. (d) is obtained from (c) by removing two nodes in its tree representation simultaneously. (e) is the result produced by `Vulcan`.

can be automatically further reduced to a smaller program, it might be easier and faster for the compiler developers to debug and pinpoint the root cause.

Figure 3.1b: Progress made by replacing an identifier. In Figure 3.1a, if we replace `l_790` on line 28 with `g_100`, the resulting program can still trigger the same crash bug. More importantly, after this change the assignment statements marked in red (lines 19–21 and 24) can be all eliminated by Perses. Figure 3.1b shows the further reduced program.

Without this change, Perses cannot eliminate these assignments because deleting them breaks the assignment chain which passes the value of `g_100` to `l_790`, where `l_790` is used as the predicate of the `if` statement on line 28. It turns out that if we delete the statements on lines 19–21 and 24 without replacing `l_790`, then `l_790` becomes uninitialized and the bug cannot be triggered. However, with this replacement, these statements become redundant as `l_790` is not used, and thus are eliminated by Perses.

Figure 3.1c: Progress made by replacing a sub-tree. In the program in Figure 3.1b, if the type specifier `int16_t` marked in orange (a sub-tree in the view of tree representation) is replaced with a different one (*e.g.*, `short`), the `typedef` statement on line 2 can be eliminated without losing the property as the only usage of the defined alias is removed. By applying Perses on the program after the replacement, a smaller program shown as Figure 3.1c is obtained. Without this replacement, Perses cannot further reduce the program in Figure 3.1b, because Perses performs only deletion-based program transformation.

Figure 3.1d: Progress made by deleting more than one node simultaneously. The 1-minimality that Perses pursues only promises that deleting a single node in the tree representation of the program will definitely make the program lose the property. Therefore if more than one node is deleted simultaneously, it is still possible for the property to be preserved. In the 1-minimal program shown in Figure 3.1c, deleting either the node representing the `for` statement or the one representing the keyword `else` makes the program lose the property due to 1-minimality. However, if these two nodes are deleted together, the property is preserved.

Figure 3.1e: Final Result of Vulcan. Figure 3.1e shows the result of Vulcan. It contains only 56 tokens, which is 51.72% fewer than the result of Perses. This result demonstrates that the potential of further reduction on 1-minimal results can be substantial. We strongly believe that the debugging process of language implementations can benefit from such a further minimized program with fewer bug-irrelevant elements.

3.3 Background

This section introduces the concept of program reduction, the existing program reduction approaches and the local minima proposed by them.

3.3.1 Relaxed Goal in Program Reduction

Finding the global minimal program is extremely difficult computationally. A previous study shows that even if the property test is constant-time, finding the global minimum is NP-complete as the well-known NP-complete problem, hitting set problem, can be reduced to the decision version of the global minimum finding problem [152]. In practice, to complete the reduction in a reasonable time, this goal is usually relaxed by existing reducers to local minima, *i.e.*, 1-minimality or 1-tree-minimality [226, 152, 209, 192, 92].

1-Minimality. 1-minimality is proposed by DD [226]. Specifically, a 1-minimal program p_{1-min} is a local minimal solution: (1) p_{1-min} passes the property test ψ and (2) any program variant derived from p_{1-min} by deleting a single element (*e.g.*, a token) cannot pass the property test ψ . It can be formally defined as follows:

$$p_{1-min} \in \{p | \psi(p) \wedge p \in \mathbb{P} \wedge \forall p' \in \mathbb{P}. |p| - |p'| = 1 \Rightarrow \neg\psi(p')\} \quad (3.1)$$

1-Tree-Minimality. 1-tree-minimality is a generalized version of the 1-minimality on the tree level. When the reduction is performed on the tree structure of the test input in a top-down manner, the concept of 1-minimality becomes inappropriate since the sub-trees rooted in different tree nodes may represent parts with different sizes. A 1-tree-minimal program p_{1-min} not only passes the property test ψ , but also requires any program variant that is derived from it by deleting a single sub-tree cannot pass the property test ψ . Mathematically, $p_{1-tree-min}$ is defined as follows:

$$p_{1-tree-min} \in \{p | \psi(p) \wedge p \in \mathbb{P} \wedge \forall p'. \text{simplify}(T(p), T(p')) \Rightarrow \neg\psi(p')\} \quad (3.2)$$

where $T(p)$ is the parse tree of program p , and the predicate $\text{simplify}(t, t') : \mathbb{T} \times \mathbb{T} \rightarrow \mathbb{B}$ is true only if t' can be derived from t by deleting a single sub-tree.

It should be noted that 1-minimality and 1-tree-minimality do not mean that no element or sub-tree can be further reduced from p_{1-min} or $p_{1-tree-min}$. If multiple elements or sub-trees are deleted at the same time, the derived program might still pass the property test.

3.4 Approach

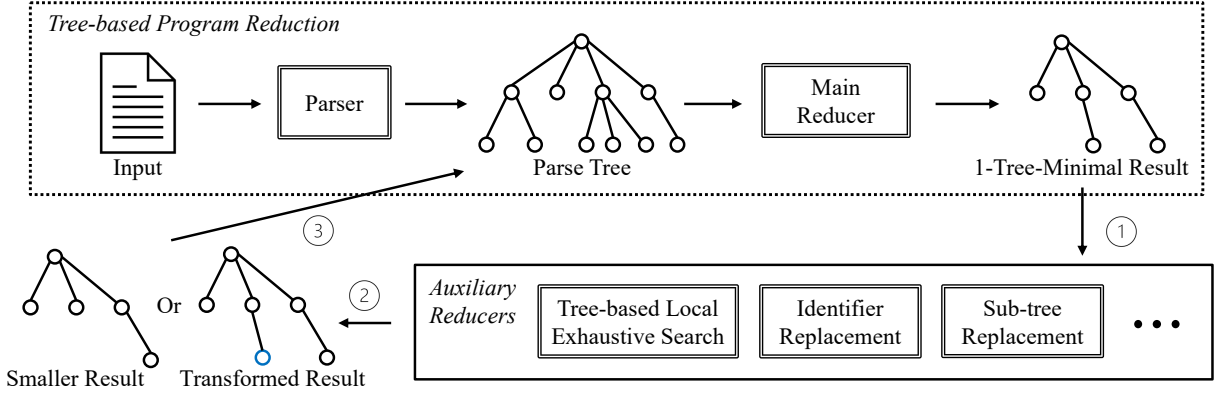


Figure 3.2: The workflow of Vulcan.

Overview. Figure 3.2 shows the workflow of Vulcan. Compared with the workflow of tree-based program reduction, Vulcan contains extra components, *i.e.*, auxiliary reducers. Given a program to be reduced, Vulcan first invokes the main reducer to produce a 1-minimal result. Then, Vulcan utilizes one of the auxiliary reducers to produce either a smaller program or a potentially non-1-minimal program by performing program transformations on the 1-minimal result (step ① and ②). This new program is then fed to the main reducer for further reduction until 1-minimality is achieved again (step ③).

The detailed workflow of Vulcan is described in Algorithm 1. It takes as input the program to be reduced P and the property checking function ψ , and eventually outputs a minimized program p such that $\psi(p)$ is true. Vulcan contains the following three stages.

Requirements on the main and auxiliary reducers. In the proposed algorithm, the main reducer must produce a 1-minimal program that either has a smaller size than the input program or is identical to the input program. For auxiliary reducers, the sizes of their outputs should either be smaller than or equal to the size of input program. Additionally, if auxiliary reducers cannot find any new program that can pass the property test, they should return the input program without any change. Conceptually, the main reducer is to quickly produce a 1-minimal result by performing aggressive, coarse-grained program transformations such as Perses and HDD, whereas the auxiliary reducers are designed to perform more fine-grained program transformations to either directly and slightly minimize the 1-minimal program or create reduction opportunities for the main reducer.

Initialization. First, Vulcan initializes a series of variables. `minimum` is the current pro-

Algorithm 1: Workflow of Vulcan

Input : P , the program to be reduced.
Input : $\psi: \mathbb{P} \rightarrow \mathbb{B}$, the property checking function
Output : A minimized program $p \in \mathbb{P}$ s.t. $\psi(p)$
Data: main_reducer: the main program reducer
Data: aux_reducers: the list of the auxiliary reducers

```
1 minimum  $\leftarrow P$ 
2 prev_by_main  $\leftarrow$  null
3  $i \leftarrow 0$ 
4 attempts  $\leftarrow 0$ 
5 while true do
    // run the main reducer to produce a 1-minimal result
6     prev_by_main  $\leftarrow$  minimum
7     minimum  $\leftarrow$  main_reducer.reduce(minimum,  $\psi$ )
8     if |minimum| < |prev_by_main| then
9         attempts  $\leftarrow 0$ 
10        prev_by_main  $\leftarrow$  minimum
    // Start running auxiliary reducers
11    while minimum == prev_by_main  $\wedge i < |aux\_reducers|$  do
12        minimum  $\leftarrow$  aux_reducers[ $i$ ].reduce(minimum,  $\psi$ )
13        attempts  $\leftarrow$  attempts + 1
14        if minimum == prev_by_main then
15             $i \leftarrow i + 1$ 
16        attempts  $\leftarrow 0$ 
17    if |minimum| < |prev_by_main| then
18        // If progress was directly made by the auxiliary reducer
19        attempts  $\leftarrow 0$ 
20    else if attempts >  $\sigma_{limit}$  then
21         $i \leftarrow i + 1$ 
22        attempts  $\leftarrow 0$ 
23    if  $i \geq |aux\_reducers|$  then return minimum
```

gram satisfying ψ . It is initialized with the program to be reduced P (line 1). `prev_by_main` is the program from which `minimum` is derived. It is initialized with `null` (line 2). i is the index of the current auxiliary reducer being used, and it is initialized to 0 (line 3). `attempts` records the number of attempts of the current auxiliary reducer. Whenever **Vulcan** switches to the next auxiliary reducer or the current auxiliary reducer helps advance the reduction progress, `attempts` is reset. It is initialized to 0 (line 4).

Reduction with the Main Reducer. After the initialization, **Vulcan** iteratively reduces `minimum` (*i.e.*, the currently smallest program) in a **while** loop (line 5-22). In each iteration, **Vulcan** first utilizes the main reducer to obtain a 1-minimal program. Specifically, the current `minimum` is first saved to `prev_by_main` (line 6), and then the main reducer is invoked to produce a new `minimum` by reducing the current `minimum` (line 7). This new `minimum` should either be smaller than or exactly the same as `prev_by_main`, and more importantly, it should always be 1-minimal. If the new `minimum` is smaller than `prev_by_main`, which means reduction progress is advanced by the main reducer, `attempts` is reset (line 9) and `prev_by_main` is updated with `minimum` again. (line 10).

Further Reduction with Auxiliary Reducers. Once a 1-minimal result is produced by the main reducer, **Vulcan** utilizes auxiliary reducers in an inner **while** loop to produce either a smaller program or a program that can be potentially reduced by the main reducer (line 11-16). In this inner loop, a new program is first produced by the currently selected auxiliary reducer (line 12), and `attempts` is incremented by one (line 13). As long as the new `minimum` is different from `prev_by_main`, this inner **while** loop for auxiliary reducers is terminated. Otherwise, if the auxiliary reducer cannot find any other program that can pass the property test, and thus it produces a program that is identical to its input (line 14), **Vulcan** switches to the next auxiliary reducer by incrementing i (line 15) and resets the `attempts` to 0 (line 16). After the increment, if i is no longer smaller than the number of auxiliary reducers (*i.e.*, there is no more auxiliary reducer to use), **Vulcan** terminates the **while** loop, and eventually returns the current `minimum` as the final result on line 22. Otherwise, **Vulcan** repeats the same process with the next selected auxiliary reducer. The size of the new `minimum` produced by the auxiliary reducers should either be smaller than or equal to the size of `prev_by_main`.

- If `minimum` is smaller than `prev_by_main` (*i.e.*, progress has been directly made by the auxiliary reducer) (line 17), `attempts` is reset (line 18), and **Vulcan** invokes the main reducer again to reduce this smaller program (line 6-10).
- If `minimum` is equal in size to `prev_by_main` (*i.e.*, progress can potentially be made by the main reducer from this new `minimum`), **Vulcan** first checks the value of `attempts`.

If **attempts** is larger than a pre-set limit σ_{limit} , **Vulcan** switches to the next auxiliary reducer (line 20), resets the **attempts** to 0 (line 21), and terminates if all the auxiliary reducers have been used (line 22). Otherwise, **Vulcan** invokes the main reducer again to reduce the new program produced by the auxiliary reducer (line 6-10).

Vulcan can work with an arbitrary number of auxiliary reducers, as long as they satisfy the aforementioned requirements. In this chapter, we proposed three simple yet effective program transformations. In the following sections, we will introduce them in detail.

3.4.1 Identifier Replacement

This program transformation is designed to minimize the number of unique identifiers and the size of the program. Specifically, *Identifier Replacement* replaces the usages of one identifier in the programs by others. By doing so, *Identifier Replacement* attempts to make the identifier unused in the program. If the program still preserves the property after this replacement, it implies that this identifier is not necessary for triggering the bug, and the definition or initialization of the unused identifier can be removed in the subsequent reduction. Although *Identifier Replacement* does not reduce the size of the program directly, the result it produces is likely to be further reduced by the tree-based reduction algorithm via removing the unused identifiers.

Algorithm 2 details the algorithm of *Identifier Replacement*. The transformation takes as input the program to be transformed and the property checking function. It first parses the program to get the parse tree in line 1. Then, it finds all the identifier nodes in the parse tree and clusters them according to their names in line 2, such that all the identifier nodes in the same **cluster** have the same unique identifier name. **id_clusters** is the list of all clusters and its length is equal to the number of unique identifier names in the parse tree t . Next, the algorithm traverses **id_clusters** and generates programs by renaming the identifiers in one of the clusters, *i.e.*, **cluster** (line 3-8). Specifically, for each **cluster**, the algorithm traverses all the other clusters in **id_clusters** and replaces all the identifiers in **cluster** except the first one with the identifier name of the other selected cluster, *i.e.*, **anotherCluster**, as shown in line 5-6. The first appearance is not replaced since in most programming languages the first appearance of an identifier is usually a definition or initialization, and the following appearances are usually the usages of this identifier. After each replacement, the algorithm checks whether the property is preserved. If so, the modified program is returned (line 7-8).

Computational Complexity Analysis. To analyze the computational complexity, we can focus on the number of property tests required by the auxiliary reducer. This is

Algorithm 2: Identifier Replacement

Input : P , the program to be transformed.

Input : $\psi: \mathbb{P} \rightarrow \mathbb{B}$, the property checking function

Output : A transformed program p , s.t. $\psi(p)$

```
1  $t \leftarrow \text{parseTree}(P)$ 
2  $\text{id\_clusters} \leftarrow$  find all the identifier nodes in  $t$  and cluster them by their name.
3 foreach  $\text{cluster} \in \text{id\_clusters}$  do
4   foreach  $\text{another\_cluster} \in \text{id\_clusters} \setminus \{ \text{cluster} \}$  do
5      $\text{id\_name} \leftarrow$  the identifier name of  $\text{another\_cluster}$ 
6      $t' \leftarrow$  a new parse tree by replacing the identifiers in  $\text{cluster}$  except the first one
       with  $\text{id\_name}$ 
7      $p \leftarrow$  program derived from  $t'$ 
8     if  $\psi(p)$  then return  $p$ 
9 return  $P$ 
```

because, during the reduction process, most time is spent on the executions of property tests [192]. Given a program, the number of property tests directly performed by *Identifier Replacement* is at most $n_c(n_c - 1)$ where n_c is the number of clusters (sets of identifiers that have the same name). If it succeeds in finding a program passing the property test, the main reducer needs to be invoked subsequently. The property tests required by the tree-based reduction algorithm may vary based on which specific algorithm is used in the framework. In practice, the computational complexity of HDD is $O(n^2)$ at worst and $O(n)$ in typical cases approximately where n represents the size of the parse tree [152]. Perses has the same worst computational complexity though it requires fewer property tests on average than HDD [192]. If the main reducer fails to reduce the size, *Identifier Replacement* will produce another program and send it to the main reducer again. This process is continued until the 1-minimal result is successfully reduced or the pre-set limit of failed attempts is reached or *Identifier Replacement* can no longer find programs that can pass the property test. Consequently, let t be the number of attempts, which is limited to a small value by a pre-set parameter, the worst computational complexity of *Identifier Replacement* can be represented as follows:

$$\mathcal{O}(t(n_c(n_c - 1) + n^2)) = \mathcal{O}(n_c^2 + n^2) = \mathcal{O}(n^2) \quad (3.3)$$

This result proves that *Vulcan* has the same worst computational complexity as HDD and Perses.

3.4.2 Sub-Tree Replacement

This transformation is inspired by mutation-based compiler fuzzing [117, 119, 189, 13, 63], a testing technique that randomly generates test programs by mutating existing programs and makes the target compiler compile these generated programs in the hope that some unexpected behavior is triggered, thus detecting bugs. Replacing a sub-tree of the program’s parse tree with a new one is a typical mutation operation applied in many existing studies [13, 210]. During a fuzzing process, it is common that different generated test programs trigger the same bug. Therefore, we believe that by replacing a sub-tree of a bug-triggering program, it is likely that a different program that triggers the same bug can be obtained, and this different program might either be smaller or bring potential reduction opportunities. To sum up, *Sub-Tree Replacement* is designed to find different programs that trigger the same bug, as it happens in the compiler fuzzing process.

Before describing the transformation, we first introduce some concepts and knowledge about parse trees and formal grammars. Each non-leaf node of a parse tree corresponds to a non-terminal symbol in the grammar. Sub-trees whose roots correspond to the same non-terminal symbol are syntactically compatible structures. In a language grammar, some non-terminal symbols have multiple production rules, which means they can have different forms. For example, a non-terminal symbol *typeSpecifier* can be `int`, `char`, or any other type, and a symbol *statement* may refer to a *if statement* or *while statement*. We refer to these different forms of a non-terminal symbol as its *alternatives*. Replacing a sub-tree with a different alternative of the non-terminal symbol corresponding to the sub-tree’s root node by the root does not break syntactical validity. Given a grammar and a specific non-terminal, a sub-tree can be generated by keeping selecting and applying production rules from the given non-terminal symbol until only terminal symbols are left [13, 115].

In summary, *Sub-Tree Replacement* systematically replaces a sub-tree with a different alternative (*e.g.*, replacing an `int` with a `char`). Algorithm 3 describes how this transformation works in detail. First, all the nodes that correspond to non-terminal symbols with multiple production rules are found (line 2). Next, the algorithm traverses all these nodes (line 3). For each node, a new sub-tree is generated from the same non-terminal as the current node to replace the original sub-tree (line 4-5). Same as the last transformation we introduced, if the modified parse tree passes the property test, the program derived by this parse tree is returned (line 7). To narrow down the search space and make generation efficient, we generate only new sub-trees that are not larger than the original sub-tree and replace only sub-trees whose non-terminal symbols represent low-level structures with a relatively small number of alternatives.

Computational Complexity Analysis. Compared to *Identifier Replacement*, *Sub-*

Algorithm 3: Sub-Tree Replacement

Input : P , the program to be transformed.

Input : $\psi: \mathbb{P} \rightarrow \mathbb{B}$, the property checking function

Output : A transformed program p , s.t. $\psi(p)$

```
1  $t \leftarrow \text{parseTree}(P)$ 
2  $\text{alternative\_nodes} \leftarrow$  All the nodes that correspond to non-terminal symbols with
   multiple production rules in  $t$ 
3 foreach  $\text{node} \in \text{alternative\_nodes}$  do
4    $st \leftarrow$  generate a sub-tree rooted in the same non-terminal as  $\text{node}$ 
5    $t' \leftarrow$  replace sub-tree rooted in  $\text{node}$  with  $st$  in  $t$ 
6    $p \leftarrow$  program derived from  $t'$ 
7   if  $\psi(p)$  then return  $p$ 
8 return  $P$ 
```

Tree Replacement provides more possibilities in terms of creating reduction opportunities. The possible programs to which it can transfer are usually more than those of *Identifier Replacement*, but the chance that the main reducer can continue to reduce elements from these new programs is lower as it lacks the heuristic that powers *Identifier Replacement*. Although, theoretically, this auxiliary reducer can directly reduce the size of the program by replacing a sub-tree with a smaller alternative, it often does not happen in practice as we limit it to replace only a sub-tree whose non-terminal represents a simple structure. Even if it produces a smaller program, the size it reduces is very limited. So, the main way this auxiliary reducer enables further reduction is by converting the input 1-minimal program to a non-1-minimal program, which enables the main reducer to further reduce the size of the program.

Suppose on average, each node in the tree has n_a alternatives, and n is the size of the parse tree; then the number of property tests directly performed by *Sub-Tree Replacement* is at most $n \times n_a$. Given that not every node has alternatives, and we restrict the transformation to target only non-terminals with a relatively small number of alternatives, n_a is smaller than n in most cases. Let t be the number of attempts, the worst computational complexity is as follows:

$$\mathcal{O}(t(n \times n_a + n^2)) = \mathcal{O}(n^2) \quad (3.4)$$

Therefore, *Sub-Tree Replacement* has the same worst-case computational complexity as HDD and Perses. But *Sub-Tree Replacement* often requires more property tests than *Identifier Replacement* to enable further reduction, as programs produced by this auxiliary reducer are less likely to be reduced by the main reducer, which means that the t required

for enabling further reduction is often larger with this auxiliary reducer.

3.4.3 Tree-Based Local Exhaustive Enumeration

In this section, we propose a deletion-based program transformation named *Tree-Based Local Exhaustive Enumeration* which uses a different deletion strategy from those that have been applied in previous AGRs. The general idea of *Tree-Based Local Exhaustive Enumeration* is to maintain a sliding window and make it slide through each level of the tree representation of the program to be reduced. For each position of the sliding window, the algorithm enumerates all the programs that can be derived by deleting nodes in the window, and then performs property tests on them.

Algorithm 4 details this algorithm. At the beginning, `node_sequence` is initialized with the highest level of the parse tree containing not fewer than σ_{wsize} nodes, where σ_{wsize} is the pre-set sliding window size (equal to 4 by default in our implementation) (line 2-4). After that, the algorithm starts another **while** loop to slide the window through `node_sequence`. At the beginning of the loop, the position of the sliding window is set to 0, *i.e.*, to include first σ_{wsize} nodes in the `node_sequence` (line 6). Next, for each position of the sliding window, the algorithm enumerates all the proper subsets of the `node_set` (*i.e.*, the set consisting of nodes in the sliding window) (line 8-9). As 1-minimality promises removing any single node breaks the property, the algorithm only needs to search for parse trees that are derived by deleting at least two nodes from t . Therefore, the proper subsets with only one element missing are filtered out (line 10).

For each remaining proper subset s , a smaller parse tree is generated by removing those nodes that are in `node_set` but not in s (line 11-14), and the program derived from this smaller parse tree is tested against the property (line 15). If the property test is passed, the algorithm does not terminate immediately because there might be more nodes that can be deleted. Instead, it sets the flag `reduced` to `true` (line 16), updates the parse tree to be reduced with the smaller parse tree (line 17), removes the deleted nodes from `node_sequence`, and finally repeats the enumeration with the sliding window staying at the same position (line 18). If the property test fails and the sliding window has not reached the end, the `window_position` is incremented by one (line 19), and the enumeration is repeated. When the sliding window reaches the end of `node_sequence`, `node_sequence` is updated to contain nodes in the next level (line 21). If the current level is the last level, the algorithm terminates and returns the reduced program p (line 20-20).

Figure 3.3 illustrates how the sliding window moves in *Tree-Based Local Exhaustive Enumeration* with a concrete example. At the beginning, there is only one node which is

the root of the parse tree in the node sequence. When the number of nodes is fewer than the length of the sliding window, the algorithm updates the node sequence with nodes in the next level (*i.e.*, replace all the non-leaf nodes in the node sequence with their children). In this case, the root has four children, and the length of the sliding window is three, so we update the node sequence to contain the four children of the root and make the initial sliding window include the first three nodes in the node sequence. When the sliding window has not reached the end of the node sequence, it moves forward by 1 node each time (step ① and ③ in Figure 3.3). Otherwise, the node sequence gets updated by replacing all non-leaf nodes with their children again (step ② in Figure 3.3) and the sliding window is again moved to the beginning of the node sequence.

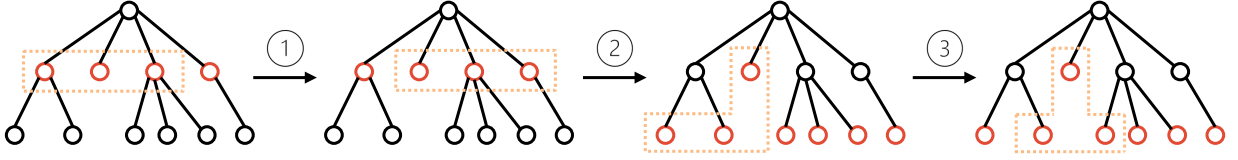


Figure 3.3: An illustrative example that shows how the sliding window (marked in orange box, length = 3) moves during the process of tree-based local exhaustive search. If the window has not reached the end of the current node sequence (marked with red), it moves forward by one node (step ① and ③). Otherwise, the node sequence is updated by replacing all non-leaf nodes with their children (step ②).

Computational Complexity Analysis. Different from *Identifier Replacement* and *Sub-Tree Replacement*, *Tree-Based Local Exhaustive Enumeration* is designed to directly reduce elements from the program. The computational complexity of this auxiliary reducer depends on the size of the tree and the window size. Suppose the parse tree to be reduced contains n nodes with a constant branching factor b , and the depth of the tree is h . Also, let the window size be equal to w . Then, in the worst case, the number of property tests required by this auxiliary reducer equals to $\sum_{i=1}^h \max(b^i - w, 0) * (2^w - w)$, which can be considered as $\mathcal{O}(n)$ at worst when w is limited to a small value. For the previous two auxiliary reducers, it might take several rounds of alternate invocation of the auxiliary reducer and the main reducer to eventually make progress on further reduction. However, for *Tree-Based Local Exhaustive Enumeration*, multiple executions are meaningless, and *Vulcan* will immediately switch to the next auxiliary reducer or terminate once the invocation of *Tree-Based Local Exhaustive Enumeration* does not make progress.

3.4.4 Order of Auxiliary Reducers

The order of auxiliary reducers may affect the efficiency and effectiveness of the reduction process. If the size of the program to be reduced can be quickly reduced by the very first auxiliary reducer, the computational cost of the following reducer invocations can be reduced, thus accelerating the reduction process. Moreover, different sequences of auxiliary reducers may lead to different results, because the previous auxiliary reducers may create or kill potential reduction opportunities, which can be caught by the subsequent auxiliary reducer.

It is non-trivial to deduce an optimal order that can achieve the best efficiency and effectiveness. First, it is hard to predict precisely how many elements each auxiliary reducer can reduce per unit time on a specific 1-minimal result before we run them. Second, without conducting extensive experiments, the effect of the order on the size of the results can hardly be analyzed.

To propose a relatively reasonable order, we recall the design goal and computational complexity of each auxiliary reducer. Among all the auxiliary reducers, *Tree-Based Local Exhaustive Enumeration* is designed to directly remove elements from the program. Also, different from other reducers, *Tree-Based Local Exhaustive Enumeration* will be switched out once it cannot produce a smaller result, which does not waste too much time even if it cannot help the reduction. Due to these reasons, we set *Tree-Based Local Exhaustive Enumeration* as the first auxiliary reducer. As for *Identifier Replacement* and *Sub-Tree Replacement*, we put the former in front of the latter, because *Sub-Tree Replacement* is more blind, and *Identifier Replacement* tends to require fewer rounds of alternate executions between the auxiliary reducer and the main reducer as the programs generated by this transformation are more likely to be non-1-minimal.

3.4.5 Parameters and Fixpoint Mode

There are two parameters that can be adjusted to control the behavior of the auxiliary reducers. For auxiliary reducers that perform non-deletion-based program transformations, the parameter σ_{limit} limits the maximum number of attempts an auxiliary reducer can make before successfully further reducing the 1-minimal result. If the auxiliary reducer continuously produces σ_{limit} programs and none of them is either smaller or able to be further reduced by the main reducer, *Vulcan* will switch to the next auxiliary reducer. For the auxiliary reducer *Tree-Based Local Exhaustive Enumeration*, the parameter σ_{wsize} decides the size of the sliding window. Both parameters can affect the performance of

Vulcan. Within a certain range, when σ_{limit} increases, the result produced by **Vulcan** will tend to be smaller since auxiliary reducers which perform non-deletion-based program transformations will try more times to search for reduction opportunities. However, the execution time of **Vulcan** will also be increased due to the same reason. Similarly, when σ_{wsize} increases, the number of programs that *Tree-Based Local Exhaustive Enumeration* generates will increase, which can potentially produce smaller results at the cost of longer execution times.

In program reduction, it is common to perform the reduction algorithm multiple times to achieve smaller results. This is possible because the deletion of some elements may enable other elements to be deleted. AGRs including HDD and Perses run in a so-called fixpoint mode to promise 1-minimality [152, 192]. In this mode, the reduction algorithm is repeatedly performed until the program can no longer be reduced (*i.e.*, reaching a fixed point). The main reducer of **Vulcan** should always run in fixpoint mode to produce a 1-minimal result. In addition, **Vulcan** can also run its auxiliary reducers in fixpoint mode. By default, **Vulcan** terminates after all the auxiliary reducers have been used. However, it is possible that the first auxiliary reducer can continue to help further reduction after the program is reduced with the help of the subsequent auxiliary reducers. Therefore, by using auxiliary reducers iteratively rather than using them only once, **Vulcan** can trade off the execution time for a potentially smaller result.

By default, **Vulcan** does not run its auxiliary reducers in fixpoint mode for less computational cost. The default values of the two parameters are set based on intuition and analysis to achieve a balance between the effectiveness and execution time. Specifically, for σ_{limit} , we tried three different values: 10, 50, and 100. The results show that when σ_{limit} equals 10, the number of queries that **Vulcan** makes is considerable. Meanwhile, the execution time is in an affordable range. On the other hand, when σ_{limit} equals 50 or 100, the experiment cannot finish within a reasonable time. Therefore, we set σ_{limit} to 10 by default.

For σ_{wsize} , we choose its default value by roughly analyzing the number of queries required when performing *Tree-Based Local Exhaustive Enumeration* with different values of σ_{wsize} . Specifically, for each value of σ_{wsize} , we calculated the number of queries required for each position of the sliding window and the lowest levels of the parse trees derived from programs of different sizes. As shown in Table 3.1, the number of queries required increases exponentially as σ_{wsize} grows, which makes the affordable choices of this parameter very limited. We think setting σ_{wsize} to 2 or 3 is too conservative. 4 or 5 seems to be the balanced choice for σ_{wsize} , and when the value of σ_{wsize} is larger than 6, the runtime overhead becomes unaffordable. Based on the above analysis, we finally set σ_{wsize} to 4 as the default value mainly for efficiency reasons.

However, users can choose to run **Vulcan** in fixpoint mode with different values of the parameters to trade off between the size of the output and execution time. To investigate the impact of the parameters and fixpoint mode and provide some insights on how to configure **Vulcan**, we evaluated a variant of **Vulcan** which uses a more aggressive setting (*i.e.*, running in fixpoint mode with larger σ_{limit} and σ_{wsize}) and conducted a parameter study. Both of the experiments will be detailed in the subsequent sections.

3.5 Evaluation

To evaluate the performance of **Vulcan**, we conducted the following research questions.

RQ1: Can **Vulcan** effectively enable further reduction from the 1-minimal result produced by AGRs?

To answer this question, we measured the effectiveness of **Vulcan** on the benchmark suite used in Perses [192], which contains 20 real-world complex bug-triggering C programs generated by fuzzing techniques, including CSmith [223] and EMI [117], and compared it with two state-of-the-art tools, Perses and C-Reduce.

RQ2: Can **Vulcan** further reduce the results output by SPRs?

To understand whether **Vulcan** is also useful for language-specific program reducers, we utilized **Vulcan** to reduce C-Reduce’s results on the C program benchmarks and measure its performance.

RQ3: Can **Vulcan** effectively reduce programs in different programming languages?

To demonstrate the generality of **Vulcan**, we measured the performance of **Vulcan** in two languages other than C/C++, *i.e.*, Rust and SMT-LIBv2. We collected 13 Rust programs by ourselves and 93 SMT-LIBv2 programs from ddSMT2.0 [114]. We compared the performance of **Vulcan** with Perses, the state-of-the-art AGR.

RQ4: What is the effectiveness of each proposed auxiliary reducer?

We conducted an ablation study to investigate to what extent each proposed auxiliary reducer contributes to the further reduction on 1-minimal result. Specifically, we implemented three variants of **Vulcan** by enabling only one of the three auxiliary reducers and evaluated their performance with the C benchmarks in RQ1.

RQ5: Can **Vulcan** output smaller results if it runs in fixpoint mode with larger σ_{limit} and σ_{wsize} ?

Finally, we investigated the performance of **Vulcan** using a more aggressive reduction setting than the default one. Concretely, we implemented another variant of **Vulcan** such that it runs reduction in fixpoint mode with larger σ_{limit} and σ_{wsize} than the default, and we evaluated this variant on the C program benchmarks.

All the experiments are run on an Ubuntu 20.04 server with Intel Xeon Gold 5217 CPU@3.00 GHz and 384 GB RAM. Every reducer is run with a single thread in all experiments.

3.5.1 RQ1: Further Reduction on 1-minimal Results

To answer this research question, we utilized **Vulcan**, **Perses** and **C-Reduce** to reduce C programs and compared their performance. These programs are from the benchmarks provided by **Perses**. In the evaluation, we measure the performance of **Vulcan** using the following metrics.

1. **Size** The number of tokens in the program produced by **Vulcan**.
2. **Queries** The number of property-testing queries issued by **Vulcan**.
3. **Time** The time spent by **Vulcan** in reduction.
4. **Percentage Change in Size** This metric is denoted as $C(\%)$ and it measures the percentage change in size of **Vulcan**'s result *w.r.t.* a baseline. Specifically, it is calculated as $\frac{|Vulcan| - |baseline|}{|baseline|} \times 100\%$, where $|Vulcan|$ and $|baseline|$ refer to the size of the results produced by **Vulcan** and a baseline, respectively.

Table 3.2 shows the reduction results. In general, **Vulcan** succeeds in producing smaller results than **Perses** for every single subject. On average, the results of **Vulcan** contain 33.55% fewer tokens than those of **Perses**. As for the execution time, **Vulcan** takes around 2 hours and 26 minutes to finish reducing on average, which is $2.45\times$ and $1.96\times$ of the time required by **Perses** and **C-Reduce** respectively, and the longest execution time is restricted in 6 hours and 15 minutes. Compared with **Perses**, we believe the extra computational cost required by **Vulcan** is acceptable considering that the size is further reduced.

Unsurprisingly, **Vulcan** does not outperform **C-Reduce** in terms of both size and execution time. On average, the results of **Vulcan** contain 80.54% more tokens than those of **C-Reduce**. This is expected since **C-Reduce** is specifically designed for reducing C/C++ programs and it has been constantly tuned and improved for a long time since it was first designed [172]. It should be noted that **Vulcan** is designed to help debug language

processors where there is no such a language-specific tool to reduce programs in the corresponding programming language. Nevertheless, there are still two subjects, *gcc-71626* and *gcc-70586*, in which **Vulcan** produces smaller results than C-Reduce.

By analyzing the results, we found that **Vulcan** requires fewer property tests than C-Reduce when the result of Perses (where auxiliary reducers start being invoked) is relatively small (*e.g.*, *gcc-61917*). However, when the result of Perses is large, **Vulcan** often requires more property tests than C-Reduce (*e.g.*, *gcc-70127*). We think this is because **Vulcan** does not have the transformations of C-Reduce that can catch language-specific reduction opportunities (*e.g.*, removing all the **typedef** in C/C++ program) and eliminate a considerable number of tokens in a very short time at the beginning.

RQ1: **Vulcan** can effectively enable further reduction on 1-minimal result produced by Perses. On average, the results of **Vulcan** contains 33.55% fewer tokens than those of Perses.

3.5.2 RQ2: Further Reduction on Language-Specific Tools' Results

Having demonstrated that **Vulcan** can effectively further reduce 1-minimal results produced by AGRs, we took one step further to investigate whether **Vulcan** can further reduce results produced by SPRs. Specifically, we used **Vulcan** to reduce C-Reduce's results on the 20 bug-triggering C programs.

The results are shown in Table 3.3. Overall, **Vulcan** further reduces C-Reduce's results in 18 out of 20 subjects. On average, 10.07% of the tokens are further reduced by **Vulcan** within around 22 minutes. Moreover, in 7 subjects, the percentage decrease in size *w.r.t.* C-Reduce is over 10%. In the best case *gcc-60116*, 29.76% tokens are further reduced by **Vulcan**.

To ensure that the potential improvement brought by **Vulcan** is contributed by our entire framework instead of the main reducer Perses, we also reduced C-Reduce's results with Perses. Table 3.3 shows the results. We found that Perses can only reduce 5 out of 20 subjects. In 4 out of these 5 subjects, **Vulcan**'s results are much smaller than Perses's results, and they have the same size in the remaining one case, *i.e.* *clang-23353*. Over these 5 subjects, the average percentage change in size of Perses's result *w.r.t.* C-Reduce is only 5.36%, which is much smaller than **Vulcan**. Therefore, we believe that the potential

improvement brought by **Vulcan** is contributed by our entire framework instead of the main reducer **Perses**.

RQ2: **Vulcan** can further reduce C-Reduce’s results by 10.07% on C program benchmarks on average. Therefore, we believe that it is worth trying to use **Vulcan** to reduce the outputs of language-specific program reduction tool for a smaller result.

3.5.3 RQ3: Generality of **Vulcan**

To demonstrate the generality of **Vulcan** in different programming languages, we evaluated **Vulcan** on Rust programs and SMT-LIBv2 programs. In both experiments, we use **Perses** as our baseline, as it is the state-of-the-art AGR. For Rust programs, we also add C-Reduce as a baseline. Although C-Reduce is designed for C/C++ programs, it can also reduce the program in other languages using its language-agnostic line-level and token-level reduction algorithms. According to the authors of C-Reduce, C-Reduce happens to perform well in reducing Rust programs [172]. We also tried to use C-Reduce to reduce SMT-LIBv2 programs. However, our preliminary evaluation shows that C-Reduce cannot effectively reduce most of these programs within a reasonable duration. While both **Perses** and **Vulcan** can reduce almost all the bug-triggering programs that contain hundreds of thousands of tokens to a program only containing hundreds of tokens in 1 hour, C-Reduce can only remove hundreds or even tens of tokens after 2 hours in such subjects. Therefore, C-Reduce is not included in the experiment with SMT-LIBv2 benchmarks.

Evaluation on Rust Programs

To evaluate the performance of **Vulcan** on Rust programs, we built a set of benchmarks consisting of 13 real-world bug-triggering Rust programs from the issue tracking system of Rust [178]. We then used C-Reduce, **Perses**, and **Vulcan** to reduce the original programs. The results are shown in Table 3.4.

Overall, **Vulcan** outperforms both **Perses** and C-Reduce in terms of the size of the reduced programs. On average, the result output by **Vulcan** contains 21.61% and 14.63% fewer tokens than those output by **Perses** and C-Reduce, respectively. This significant improvement is particularly evident in the case of the bug-triggering program *rust-78005*, where the result produced by **Vulcan** contains 88.24% and 84.00% fewer tokens than those produced by **Perses** and C-Reduce, respectively. Additionally, it is important to note that the most valuable feature of **Vulcan** is its ability to further reduce programs that cannot

be sufficiently reduced by previous AGRs. If we consider only cases where the programs still contain more than 50 tokens after being reduced by Perses, the percentage changes in size of **Vulcan**’s results *w.r.t.* Perses and C-Reduce are 27.95% and 19.84%, respectively. In terms of execution time, **Vulcan** can finish reduction in 2 hours for each subject, and in 10 out of 13 subjects, the reduction is finished in less than 5 minutes.

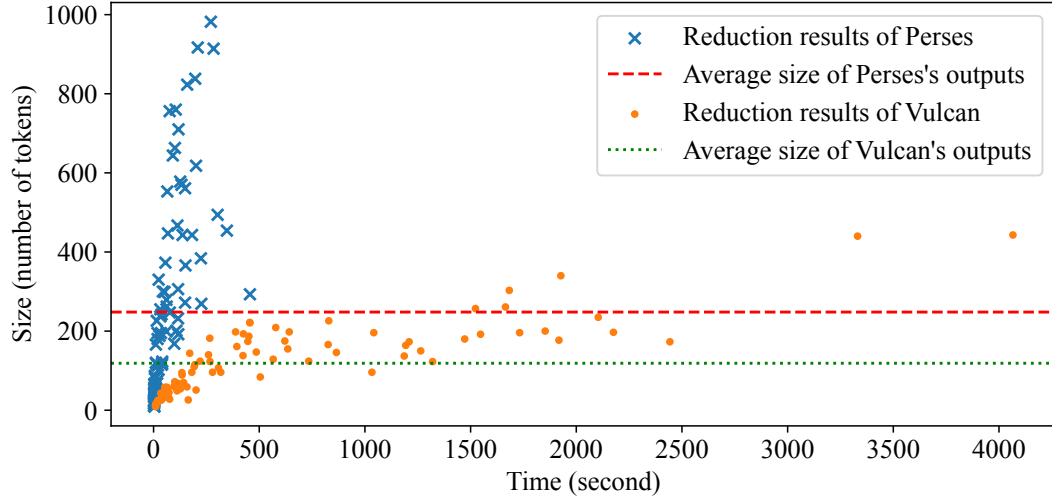
We manually analyzed the case *rust-78720* since it is the only case where C-Reduce outperforms **Vulcan** in terms of size. We found that the result of C-Reduce is syntactically invalid. In this case, the program does not have to be syntactically valid to trigger the bug. C-Reduce can reduce *rust-78720* to such a result because its token-level and line-level program reducers do not care about the syntax of the program. However, **Vulcan** mainly focuses on generating and performing property testing on syntactically valid programs to improve efficiency. Thus, it misses the small syntactically invalid result and produces a slightly larger but syntactically valid program.

Evaluation on SMT-LIBv2 Programs

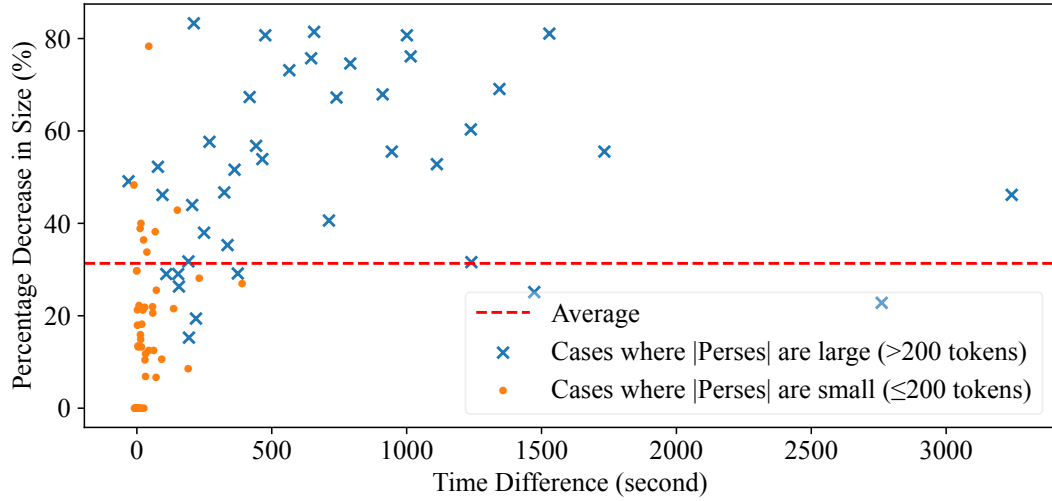
The programs we used for this experiment are from ddSMT2.0 [114]. We filtered out the programs that cannot be parsed because they cannot be handled by the baseline Perses, a tree-based program reduction tool. In total, 93 programs were collected for our evaluation and their numbers of tokens range from 15 to 120,757 (average 6,917).

Figure 3.4a shows the experiment results. Each cross or dot in the figure represents a reduction result, where its x-coordinate is the execution time of the reduction, and the y-coordinate is the size of the reduced program. To better visualize the difference between Perses’s and **Vulcan**’s results for each subject, we calculated the percentage decrease in size that **Vulcan** achieves *w.r.t.* Perses for each subject and the extra time required by **Vulcan** in reduction. The comparison results are shown in Figure 3.4b. On average, the percentage decrease in size of **Vulcan** *w.r.t.* Perses is 31.34%. In 75 out of 93 subjects, the output of **Vulcan** is smaller than that of Perses, and in 23 subjects, the size of **Vulcan**’s output is smaller than half of the size of Perses’s output. In terms of execution time, in 91 out of 93 subjects, the extra time required by **Vulcan** is less than half an hour, and in the other two subjects, **Vulcan** can finish within an extra hour.

As mentioned previously, **Vulcan** is most useful when previous AGRs cannot sufficiently reduce the program. To investigate how **Vulcan** performs in this intended application scenario, we excluded the cases where the results output by Perses contain fewer than 200 tokens, and evaluated **Vulcan** with the remaining 40 cases. In these 40 cases, the output of **Vulcan** is consistently smaller than that of Perses, and in 22 cases, the size of **Vulcan**’s



(a) Execution time and the size of reduced programs of Perses and Vulcan.



(b) The percentage decrease achieved and the extra time required by Vulcan *w.r.t.* Perses. $|\text{Perses}|$ represents the size of Perses's output.

Figure 3.4: Evaluation results of Perses and Vulcan on SMT-LIBv2 benchmarks.

output is smaller than half of the size of Perses’s output. Moreover, on average, in the intended application scenario of **Vulcan**, the percentage decrease in size is 52.01%, which further demonstrates the effectiveness of **Vulcan**.

RQ3: **Vulcan** can effectively reduce programs in different programming languages. For Rust programs, on average, **Vulcan** can produce results that contain 21.61% fewer and 14.63% fewer tokens than those of Perses and C-Reduce, respectively. For SMT-LIBv2 programs, **Vulcan**’s output contains 31.34% fewer tokens than Perses’s output on average. If we restrict our analysis to cases where Perses cannot sufficiently reduce the program, the calculated average percentage decreases in size presented above will be increased to 27.95%, 19.84%, and 52.01%, respectively.

3.5.4 RQ4: Effectiveness of Each Auxiliary Reducer

In this research question, we evaluated the performance of each auxiliary reducer proposed by us. Specifically, we created three variants of **Vulcan**, namely, **Vulcan_{EE}**, **Vulcan_{IR}**, and **Vulcan_{SR}** where each of them only enables one of the three proposed auxiliary reducers: *Tree-Based Local Exhaustive Enumeration* (EE), *Sub-Tree Replacement* (SR), and *Identifier Replacement* (IR), respectively. To focus on the performance of each auxiliary reducer, we run these variants with the 1-minimal results produced by Perses on the C program benchmarks. This method helps us to save property test queries and execution time required to achieve 1-minimality.

Table 3.5 shows the results of all the three variants reducing the C programs that have been reduced by Perses. In general, all three variants can reduce the size of Perses’s results in some subjects of the C program benchmarks. Specifically, **Vulcan_{EE}** has the shortest execution time and it succeeds in reducing tokens in 17 out of 20 subjects. **Vulcan_{IR}** succeeds in reducing tokens in all subjects, and it reduces around 26% of the tokens in Perses’s results on average. **Vulcan_{SR}** reduces tokens in 8 out of 20 subjects, which is not as good as the other two variants but still effective. Interestingly, in four subjects, **Vulcan_{IR}** eliminates more tokens than **Vulcan**. We believe this is because **Vulcan** utilizes *Tree-Based Local Exhaustive Enumeration* before *Identifier Replacement*, and the effect of *Tree-Based Local Exhaustive Enumeration* kills some better potential reduction opportunities which originally could be caught by *Identifier Replacement*.

We also measured the reduction speed of each variant, *i.e.*, the number of eliminated tokens per second. On average, **Vulcan_{EE}** can reduce 0.029 tokens per second, which is the fastest one among all the three variants. The second fastest one is **Vulcan_{IR}**, which can

reduce 0.014 tokens per second. `VulcanSR` is the slowest, which reduces only 0.005 tokens per second. This result conforms to our analytical result introduced in § 3.4.4.

RQ4: All the proposed auxiliary reducers are effective in enabling further reduction from 1-minimal results produced by AGRs. Among them, *Identifier Replacement* is the most effective one, and *Tree-Based Local Exhaustive Enumeration* is the most efficient one.

3.5.5 RQ5: Performance with a More Aggressive Reduction Setting

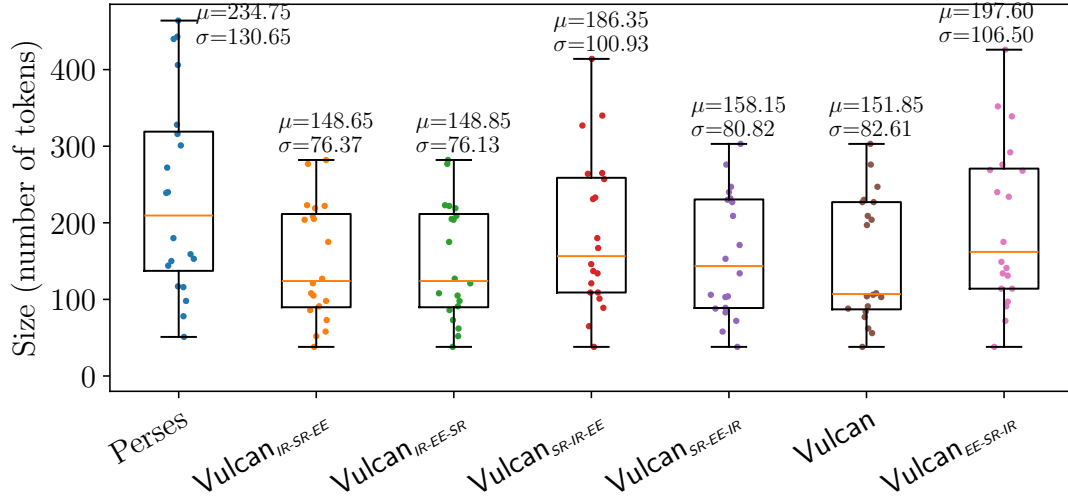
In this research question, we investigated whether `Vulcan` can produce a smaller result if we increase the value of two parameters, σ_{limit} and σ_{wsize} , and run `Vulcan` in fixpoint mode. We implemented a variant of `Vulcan` named `Vulcan+` by increasing σ_{limit} from 10 to 15 and σ_{wsize} from 4 to 6, and also making it run in fixpoint mode.

We evaluated `Vulcan+` on the same C programs in RQ1. The results are shown in Table 3.6. Overall, the reduced program of `Vulcan+` contains 5.89% fewer tokens than that of `Vulcan` at the cost of requiring around $1.8\times$ execution time. In 14 out of 20 subjects, `Vulcan+` succeeds in producing a smaller result than Perses. Notably, in 7 subjects, the reduced program of `Vulcan+` is 10% smaller than that of `Vulcan`, and in the subject *clang-23309*, this percentage reaches 27.06%.

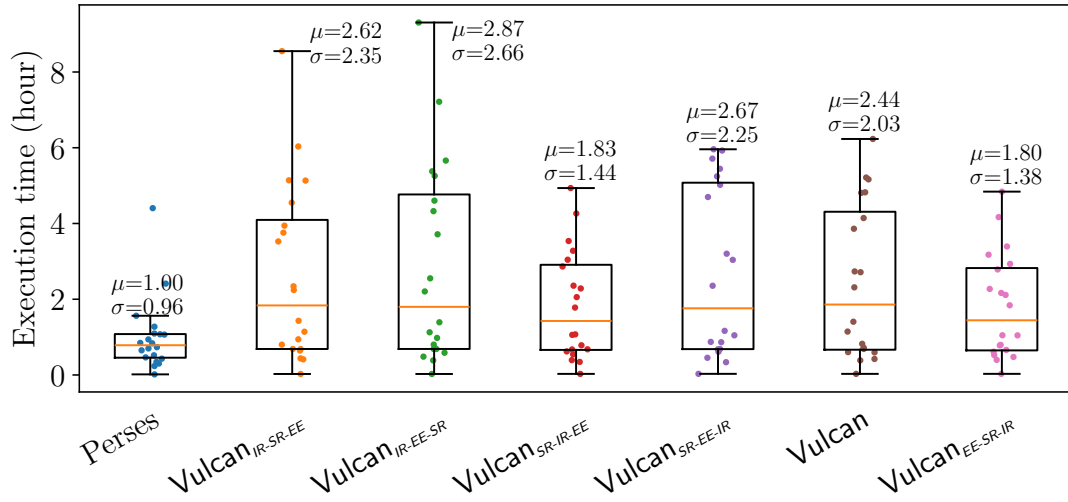
RQ5: On average, the output of `Vulcan+` contains 5.89% fewer tokens than that of `Vulcan` at the cost of taking $1.81 \times$ the execution time of `Vulcan` to finish, which demonstrates `Vulcan` can produce smaller results if it is deployed with a more aggressive setting.

3.6 Discussion

In this section, we discuss how the values of σ_{limit} and σ_{wsize} and the orders of the auxiliary reducers affect the performance of `Vulcan`.



(a) Sizes of the reduced programs.



(b) Execution time.

Figure 3.5: Evaluation results of Vulcan using different orders of auxiliary reducers on the C benchmarks.

3.6.1 Impact of the Order of the Auxiliary Reducers

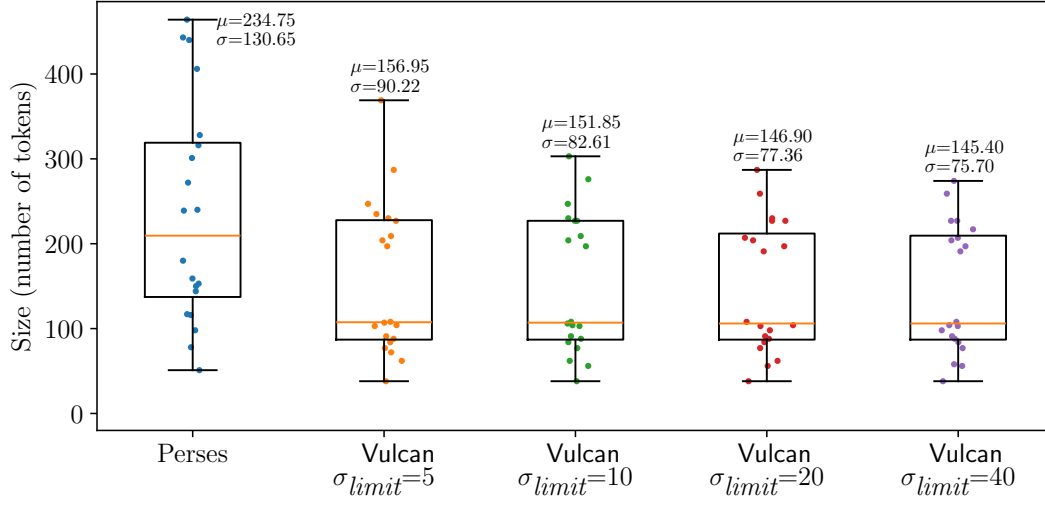
As aforementioned in § 3.4.4, we deduce a relatively reasonable order by analyzing the design goal and computational complexity of each auxiliary reducer and set this order as the default order of **Vulcan**. In § 3.5, the extensive evaluation results demonstrated the effectiveness of this order. However, it is unclear how **Vulcan** performs with other orders of the auxiliary reducers. To better understand how the orders of the auxiliary reducers affect the performance, we proposed five variants: **Vulcan**_{IR-SR-EE}, **Vulcan**_{IR-EE-SR}, **Vulcan**_{SR-IR-EE}, **Vulcan**_{SR-EE-IR}, and **Vulcan**_{EE-SR-IR}. Each of them has a different order of the auxiliary reducers from **Vulcan**, and the subscript indicates the order. We ran **Vulcan** and all these variants to reduce the 20 C programs.

As shown in Figure 3.5, all the variants of **Vulcan** have different performance, but all of them significantly outperform **Perses** in terms of the output’s size. In general, the variant which produces smaller results overall also requires a longer execution time on average. For example, **Vulcan**_{IR-SR-EE} and **Vulcan**_{IR-EE-SR} produce smaller results in general than other variants, but they also have relatively longer execution times than other variants. Nevertheless, some variants outperformed other variants in terms of both metrics. For example, compared with **Vulcan**_{SR-EE-IR}, both **Vulcan** and **Vulcan**_{IR-SR-EE} produce smaller results while taking a shorter execution time on average. As for the default order that we proposed, its performance in terms of output’s average size is comparable to **Vulcan**_{IR-SR-EE}, the best variant on this metric. Meanwhile, the average execution time taken by the default order is relatively short, which is only longer than those of **Vulcan**_{SR-IR-EE} and **Vulcan**_{EE-SR-IR}, the two most ineffective variants.

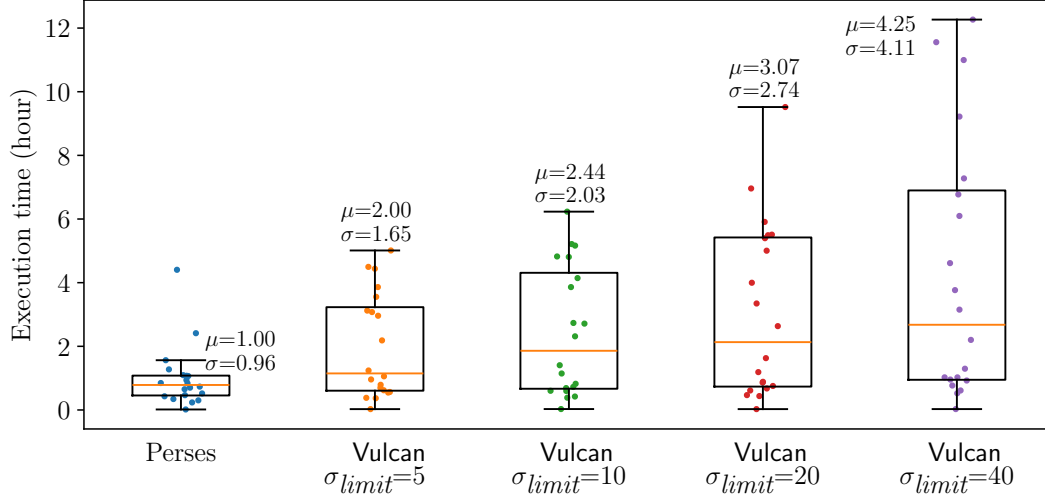
3.6.2 Impact of the Parameters

In the prototype of **Vulcan**, the two parameters σ_{limit} and σ_{wsize} are set with default values. These values are selected based on our intuition and analysis to achieve a balance between the effectiveness and execution time. However, it should be noted that these parameters can be customized by users to trade off the size of the result versus the execution time to suit their needs.

To provide more insights for adjusting the parameters, we conducted a parameter study to investigate the impact of the two parameters. In the experiment, we keep one parameter with its default value and set the other parameter with 3 different values. Figure 3.6 shows the results of running **Vulcan** on the C benchmarks with different σ_{limit} . As shown in Figure 3.6a, **Vulcan** tends to produce smaller results with a higher value of σ_{limit} , though

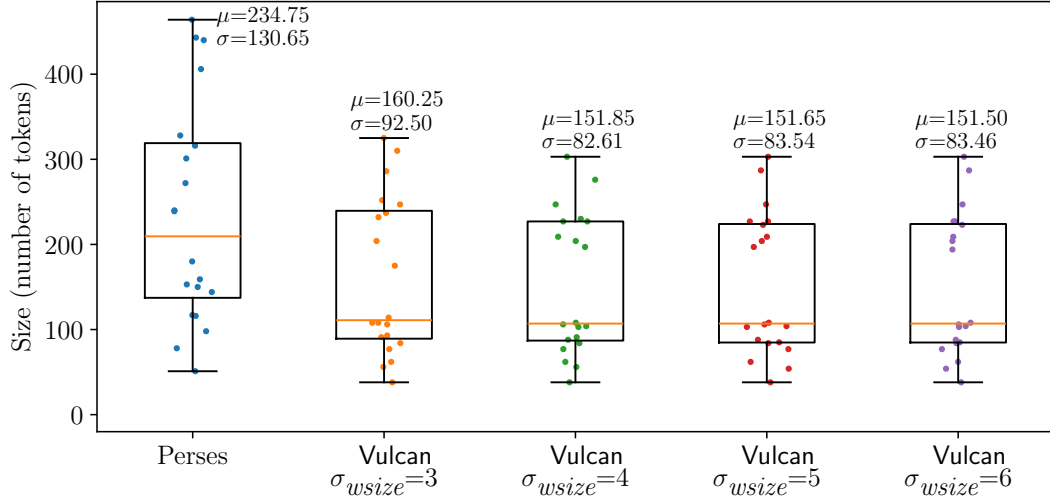


(a) Sizes of the reduced programs.

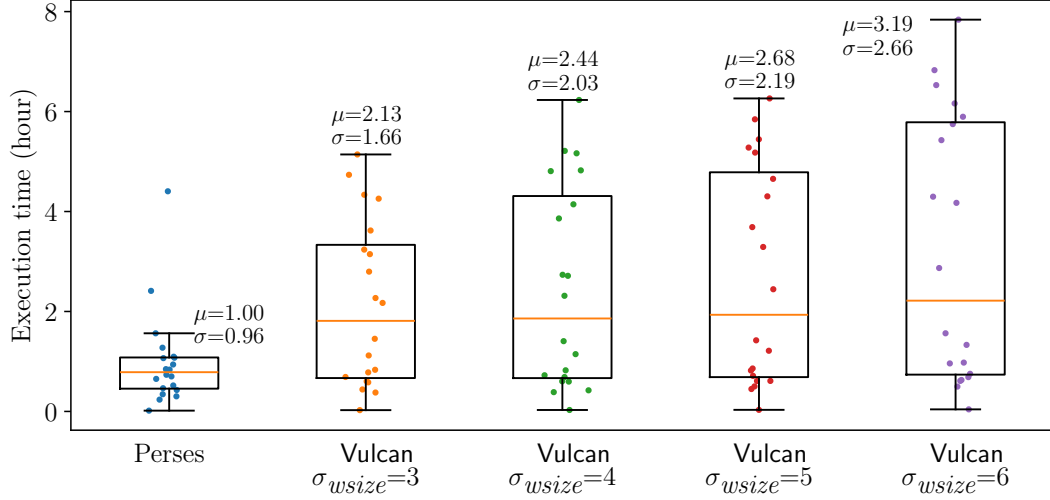


(b) Execution time.

Figure 3.6: Evaluation results of Vulcan with different σ_{limit} on the C benchmarks.



(a) Sizes of the reduced programs.



(b) Execution time.

Figure 3.7: Evaluation results of Vulcan with different σ_{wsiz} on the C benchmarks.

the improvement is moderate. As for the execution time, Figure 3.6b shows the experiment result. In general, as σ_{limit} increases, the required execution time noticeably increases.

The impact of σ_{wsize} is investigated in the same way. As shown in Figure 3.7, in general, when σ_{wsize} increases, the size of the output decreases, and the execution time increases. However, the overall decrease in size of the result is rather limited, especially when σ_{wsize} is greater than 3.

3.6.3 Threats to Validity

The correctness of the **Vulcan** implementation is essential to internal validity. To ensure that the reduction performed by **Vulcan** is correct, we verify that each reduced program still passes the corresponding property test. In addition, to prevent bug slippage (i.e., cases where the reduced program triggers a different bug), the property tests used in our benchmarks are designed to be comprehensive. For example, as mentioned in § 2.1.2, rather than checking for the presence of a bug symptom on only a single buggy compiler version, the property checker evaluates the program using multiple compilers or multiple versions of the same compiler, including both buggy and correct versions.

Another potential threat to internal validity arises from nondeterminism in property tests. For some benchmarks, the bug symptom does not manifest in every compilation. To mitigate this issue, the property-checking scripts invoke the compilation multiple times, thereby reducing the impact of nondeterministic behavior.

To address threats to external validity, we collect a large and diverse set of benchmarks to ensure that the evaluation covers both crash and miscompilation bugs, different languages, and different compilers.

3.7 Related Work

AGRs. Delta Debugging (DD) [226] is the first work that opened up the research direction of test case reduction. It proposed two algorithms, **dd** and **ddmin**. **dd** can minimize the difference between a failure-inducing test case and a given test case that can be correctly handled. **ddmin** is a special case of **dd**, and it directly minimizes the failure-inducing test case by minimizing the difference between the test case and an empty test case. However, **ddmin** regards code as plain strings of tokens, and thus it does not perform well when the failure-inducing test case is large and structural. To overcome this obstacle, Misherghi and

Su proposed Hierarchical Delta Debugging (HDD) [152], which leverages the tree structure of the test cases to facilitate the reduction process. Specifically, HDD performs `ddmin`, the algorithm proposed in DD, on the parse tree of the test case, from the topmost level to the bottommost level. In this way, HDD can efficiently handle large and highly-structured test cases. Hodován *et al.* also proposed Coarse Hierarchical Delta Debugging (CHDD) to further accelerate the reduction process [97]. Berkeley Delta utilizes *topformflat* to identify nested structures and then perform `ddmin` on them [151]. Despite their effectiveness, none of these aforementioned techniques preserves the syntactical validity of the test case during the reduction process. As a result, when these tools perform program reduction, many syntactically invalid programs are generated and tested against the property. These invalid programs can hardly pass the property tests, and executing property tests on these program wastes considerable time. To solve this problem, Sun *et al.* proposed Perses, a syntax-guided program reduction tool to facilitate the reduction process by promising only to generate and test syntactically valid program [192]. CHISEL is another program reduction tool designed for program debloating [92]. It accelerates the reduction by using a reinforcement learning-based approach. Moreover, ProbDD introduces Bayesian optimization to model the probability of each element appearing in the minimal result, and prioritizes the reduction of those with high probability [209].

These reduction techniques aforementioned strive to reduce programs to 1-minimality. Once a 1-minimal result is obtained, they cannot further reduce the program. By contrast, **Vulcan** can continue reducing by performing fine-granularity program transformations and provide developers a much smaller bug-triggering test case, thus facilitating the debugging process.

SPRs. In addition to the language-agnostic program reduction tools like HDD and Perses, some reducers focus on reducing specific languages. For example, C-Reduce is specifically designed to reduce C/C++ programs [170]. It first uses the Clang front end to parse the program to be reduced and then invokes a collection of plugins to reduce the program. Each plugin performs a series of well-crafted transformations designed only for C/C++ languages. J-Reduce is a reduction tool for Java bytecode proposed by Kalhauge and Palsberg [109]. They modeled the task of reducing inputs with many internal dependencies as a problem of dependency graph reduction and proposed a general strategy to solve this problem. Their following work further improved J-Reduce using finer-grained modeling of dependencies [110]. JS Delta employs WALA static analysis [101] to reduce JavaScript programs [108]. ddSMT is a program reduction tool for the SMT-LIBv2 format specifically [161]. It achieves better performance on inputs in SMT-LIBv2 format by supporting handling the language-specific features, such as macros, named annotations, and scopes defined by `push` and `pop` commands. ddSMT2.0 is a successor of ddSMT [114]. It

improves ddSMT by using a hybrid minimization strategy including both the **ddmin**-based strategy used in ddSMT and an orthogonal hierarchical strategy. It also extends ddSMT by supporting the entire family of SMT-LIBv2 language dialects. All these SPRs usually can work effectively for only one specific programming language. In contrast, our framework, **Vulcan**, is a language-agnostic program reduction tool and can effectively reduce programs in different languages.

3.8 Chapter Conclusion

In this chapter, we proposed **Vulcan**, a language-agnostic program reduction framework to further minimize the results of AGRs by exploiting the formal syntax of the language to perform diverse program transformations. These transformations can either create reduction opportunities for other reduction algorithms to progress or directly delete bug-irrelevant program elements from the results. We also proposed three simple but effective examples of program transformations: *Identifier Replacement*, *Sub-Tree Replacement*, and *Tree-Based Local Exhaustive Enumeration*, and implemented a prototype of **Vulcan** on top of Perses with these three transformations incorporated.

By conducting extensive evaluation with multilingual benchmarks including C, Rust and SMT-LIBv2 programs, we demonstrated the effectiveness and generality of **Vulcan**. On average, the result of **Vulcan** contains 33.55%, 21.61%, and 31.34% fewer tokens than that of the state-of-the-art AGR, Perses, on C, Rust, and SMT-LIBv2 subjects respectively. If more execution time is allocated, the result of **Vulcan** can be even smaller. Moreover, **Vulcan** can even further reduce the results produced by C-Reduce for C programs. We believe **Vulcan** can effectively facilitate the debugging process of language processors.

Algorithm 4: Tree-Based Local Exhaustive Enumeration

Input : P , the program to be transformed.
Input : $\psi: \mathbb{P} \rightarrow \mathbb{B}$, the property checking function
Output : A reduced program p , *s.t.* $\psi(p)$

```
1  $t \leftarrow \text{parseTree}(P)$ 
2  $\text{node\_sequence} \leftarrow [\text{root node of } t]$ 
3 while length of node_sequence  $< \sigma_{\text{wsize}}$  do
4    $\text{node\_sequence} \leftarrow \text{nodes in the next level of } t$ 
5 while true do
6    $\text{window\_position} \leftarrow 0$ 
7   while  $\text{window\_position} + \sigma_{\text{wsize}} < \text{length of node\_sequence}$  do
8      $\text{node\_set} \leftarrow \text{node\_sequence}[\text{window\_position} : \text{window\_position} + \sigma_{\text{wsize}}]$ 
9      $\text{proper\_subsets} \leftarrow \text{all true subsets of node\_set}$ 
10     $\text{filtered\_proper\_subsets} \leftarrow \{s \mid s \in \text{proper\_subsets} \wedge |\text{node\_set} \setminus s| \geq 2\}$ 
11    foreach  $s \in \text{filtered\_proper\_subsets}$  do
12       $\text{nodes\_to\_remove} \leftarrow \text{node\_set} \setminus s$ 
13       $t' \leftarrow \text{a new tree obtained by removing nodes in nodes\_to\_remove from } t$ 
14       $p \leftarrow \text{program derived from } t'$ 
15      if  $\psi(p)$  then
16         $\text{reduced} \leftarrow \text{true}$ 
17         $t \leftarrow t'$ 
18         $\text{remove the removed nodes from node\_sequence}$ 
19      else  $\text{window\_position} \leftarrow \text{window\_position} + 1$ 
20    if all nodes in node_sequence are leaf nodes then break
21     $\text{node\_sequence} \leftarrow \text{nodes in the next level of } t$ 
22 return  $p$ 
```

Table 3.1: Number of queries required by performing Tree-Based Local Exhaustive Enumeration with different values of σ_{wsize} . $Q_{pos}(\sigma_{wsize})$ is the number of queries required by each position of the sliding window given the σ_{wsize} . $Q_{level}(\sigma_{wsize})$ is the number of queries required by the lowest level of a parse tree that has s leaf nodes given the σ_{wsize} .

σ_{wsize}	$Q_{pos}(\sigma_{wsize})$	$Q_{level}(\sigma_{wsize})$			
		$s = 100$	$s = 200$	$s = 300$	$s = 400$
2	1	99	199	299	399
3	4	392	792	1,192	1,592
4	11	1,067	2,167	3,267	4,367
5	26	2,496	5,096	7,696	10,296
6	57	5,415	11,115	16,815	22,515
7	120	11,280	23,280	35,280	47,280
8	247	22,974	47,671	72,371	97,071

Table 3.2: Evaluation results of Perses, C-Reduce, and Vulcan on C programs. The size of a program is measure by the number of tokens it contains. O(#) is the size of the **O**riginal bug-triggering program. R(#) is the size of the **R**educed program. Q(#) is the number of required property test **Q**ueries. T(s) is the execution **T**ime in seconds. C(%) is the percentage **C**hanges in size of **V**ulcan’s results *w.r.t.* Perses and C-Reduce.

Bug	O(#)	Perses			C-Reduce			Vulcan			C(%) <i>w.r.t.</i>	
		R(#)	Q(#)	T(s)	R(#)	Q(#)	T(s)	R(#)	Q(#)	T(s)	Perses	C-Reduce
clang-22382	21,068	144	2,392	851	70	13,186	1,364	108	10,363	1,521	-25.00%	54.29%
clang-22704	184,444	78	1,828	1,870	42	11,189	2,263	62	5,181	2,171	-20.51%	47.62%
clang-23309	38,647	464	4,626	3,012	118	32,167	4,455	303	62,593	9,843	-34.70%	156.78%
clang-23353	30,196	98	2,820	1,088	74	12,299	1,410	91	7,872	1,389	-7.14%	22.97%
clang-25900	78,960	239	2,527	1,672	90	16,517	2,022	104	10,382	2,605	-56.49%	15.56%
clang-26760	209,577	116	2,215	3,380	43	12,793	2,946	56	7,204	4,126	-51.72%	30.23%
clang-27137	174,538	180	5,046	15,857	50	25,049	9,078	88	14,085	18,593	-51.11%	76.00%
clang-27747	173,840	117	1,685	1,237	68	15,138	2,223	77	8,480	2,477	-34.19%	13.24%
clang-31259	48,799	406	2,435	2,523	168	26,997	5,549	276	56,473	17,364	-32.02%	64.29%
gcc-59903	57,581	316	4,247	5,628	105	48,652	7,314	197	26,095	8,330	-37.66%	87.62%
gcc-60116	75,224	443	5,335	3,839	168	39,262	6,131	247	59,359	9,768	-44.24%	47.02%
gcc-61383	32,449	272	3,476	3,941	84	23,169	4,516	209	31,705	13,900	-23.16%	148.81%
gcc-61917	85,359	150	2,941	2,344	65	23,138	3,832	103	10,028	2,963	-31.33%	58.46%
gcc-64990	148,931	240	3,460	3,047	65	23,358	5,190	204	14,823	5,061	-15.00%	213.85%
gcc-65383	43,942	153	2,031	1,547	63	14,840	2,280	84	4,918	2,150	-45.10%	33.33%
gcc-66186	47,481	328	2,756	2,642	115	18,258	5,153	227	26,615	18,767	-30.79%	97.39%
gcc-66375	65,488	440	3,141	3,869	56	21,181	8,057	227	30,792	14,918	-48.41%	305.36%
gcc-70127	154,816	301	2,772	4,590	84	21,507	7,556	230	30,502	17,309	-23.59%	173.81%
gcc-70586	212,259	159	4,458	8,684	130	35,794	8,043	106	18,326	22,434	-33.33%	-18.46%
gcc-71626	6,133	51	575	57	46	7,179	417	38	1,763	104	-25.49%	-17.39%
median	70,356	210	2,796	2,827	72	21,344	4,486	107	14,454	6,696	-32.68%	56.37%
mean	94,487	235	3,038	3,584	85	22,084	4,490	152	21,878	8,790	-33.55%	80.54%

Table 3.3: Evaluation results of **Vulcan** and **Perses** on C-Reduce reduced C programs. C(%) is the percentage **C**hange in size *w.r.t.* the result of C-Reduce.

Bug	C-Reduce		Perses			Vulcan			
	R(#)	R(#)	Q(#)	T(s)	C(%)	R(#)	Q(#)	T(s)	C(%)
clang-22382	70	70	74	13	0	70	1,907	218	0
clang-22704	42	42	33	9	0	40	975	120	-4.76%
clang-23309	118	112	234	44	-5.08%	76	4,968	744	-35.59%
clang-23353	74	72	109	22	-2.70%	72	2,442	299	-2.70%
clang-25900	90	90	110	26	0	83	3,136	452	-7.78%
clang-26760	43	43	46	15	0	38	1,082	176	-11.63%
clang-27137	50	50	49	70	0	47	1,229	896	-6.00%
clang-27747	68	68	80	31	0	66	2,350	450	-2.94%
clang-31259	168	168	245	185	0	131	8,861	6,303	-22.02%
gcc-59903	105	105	122	52	0	103	3,360	646	-1.90%
gcc-60116	168	159	355	126	-5.36%	118	10,191	3,622	-29.76%
gcc-61383	84	84	145	79	0	84	2,409	2,179	0
gcc-61917	65	65	70	17	0	62	1,761	238	-4.62%
gcc-64990	65	65	58	19	0	62	1,580	306	-4.62%
gcc-65383	63	61	81	35	-3.17%	51	1,699	434	-19.05%
gcc-66186	115	115	294	114	0	112	3,153	3,374	-2.61%
gcc-66375	56	56	62	137	0	53	1,670	2,468	-5.36%
gcc-70127	84	84	98	117	0	76	2,614	1,521	-9.52%
gcc-70586	130	128	259	112	-1.54%	113	5,514	1,573	-13.08%
gcc-71626	46	46	42	3	0	38	1,054	37	-17.39%
median	72	71	90	40	0%	71	2,380	549	-5.68%
mean	85	84	116	73	-0.89%	75	3,098	1,303	-10.07%

Table 3.4: Evaluation results of Perses, C-Reduce, and Vulcan on Rust programs. C(%) is the percentage Changes in size of Vulcan’s results *w.r.t.* Perses and C-Reduce.

Bug	O(#)	Perses			C-Reduce			Vulcan			C(%) w.r.t	
		R(#)	Q(#)	T(s)	R(#)	Q(#)	T(s)	R(#)	Q(#)	T(s)	Perses	C-Reduce
rust-44800	801	467	1,873	1,122	471	39,781	7,735	285	29,331	7,170	-38.97%	-39.49%
rust-69039	190	114	761	523	109	7,649	734	101	17,739	5,285	-11.40%	-7.34%
rust-77002	347	263	1,221	311	264	21,470	2,276	247	13,605	1,796	-6.08%	-6.44%
rust-77320	173	39	99	13	39	2,581	166	39	959	66	0	0
rust-77323	81	13	38	4	13	453	28	13	194	13	0	0
rust-77910	63	34	104	13	23	1,415	101	21	985	74	-38.24%	-8.70%
rust-77919	132	74	297	55	70	6,190	459	62	3,807	272	-16.22%	-11.43%
rust-78005	182	102	281	20	75	5,172	344	12	708	73	-88.24%	-84.00%
rust-78325	65	28	52	15	34	2,341	130	28	494	41	0	-17.65%
rust-78651	957	17	83	36	12	1,414	86	9	258	52	-47.06%	-25.00%
rust-78652	263	56	182	39	49	3,261	195	49	3,107	198	-12.50%	0
rust-78655	28	26	39	4	26	1,813	98	26	471	27	0	0
rust-78720	121	72	171	32	51	6,700	523	56	2,934	267	-22.22%	9.8%
median	173	56	171	32	49	3,261	195	39	1,052	93	-12.50%	-7.34%
mean	262	100	400	168	95	7,711	990	73	5,738	1,180	-21.61%	-14.63%

Table 3.5: Results of Vulcan_{EE} , Vulcan_{IR} and Vulcan_{SR} reducing the 1-tree-minimal program output by Perses. $E(\#)$ is the number of **E**liminated tokens in reduction.

Bug	Perses	Vulcan _{EE}			Vulcan _{IR}			Vulcan _{SR}			Vulcan
	R(#)	E(#)	Q(#)	T(s)	E(#)	Q(#)	T(s)	E(#)	Q(#)	T(s)	E(#)
clang-22382	144	0	2,081	99	36	5,935	565	10	2,771	272	36
clang-22704	78	4	2,171	108	12	582	59	6	4,332	420	16
clang-23309	464	38	34,366	1,965	163	25,414	4,611	0	6,482	1,109	161
clang-23353	98	7	2,676	110	7	2,871	230	0	1,689	176	7
clang-25900	239	108	4,525	245	45	11,269	1,356	0	3,521	404	135
clang-26760	116	2	2,234	125	42	1,836	474	0	1,459	296	60
clang-27137	180	5	5,317	513	87	3,951	2,172	0	2,356	965	92
clang-27747	117	3	2,160	164	40	3,862	805	0	1,715	448	40
clang-31259	406	54	31,021	3,761	84	20,586	11,483	0	5,127	3,577	130
gcc-59903	316	7	9,737	564	141	12,569	2,071	126	6,692	1,394	119
gcc-60116	443	104	39,934	2,485	200	28,561	7,329	0	5,955	1,780	196
gcc-61383	272	38	19,185	2,817	35	11,660	7,479	10	4,654	4,164	63
gcc-61917	150	2	3,613	164	42	3,757	428	7	2,376	280	47
gcc-64990	240	0	3,505	193	36	7,812	1,701	0	2,947	497	36
gcc-65383	153	56	1,682	160	26	3,418	796	0	1,966	458	69
gcc-66186	328	60	9,860	1,529	48	11,637	9,063	0	4,187	3,403	101
gcc-66375	440	171	17,028	3,448	61	52,589	23,075	0	6,184	3,955	213
gcc-70127	301	25	16,428	3,592	73	12,330	9,631	0	4,085	4,464	71
gcc-70586	159	9	7,628	1,768	53	6,017	9,201	1	2,909	4,556	53
gcc-71626	51	0	643	18	10	218	10	3	699	28	13
median	210	8	8,708	3,838	44	10,874	6,186	0	6,632	3,899	66
mean	235	35	13,828	4,775	62	14,382	8,211	8	6,644	5,216	82.9

Table 3.6: Evaluation results of **Vulcan** and **Vulcan⁺** on C program benchmarks. C(%) is the size change percentages of **Vulcan⁺**.

Bug	Perses	Vulcan			Vulcan ⁺			C(%) <i>w.r.t.</i>	
	R(#)	R(#)	Q(#)	T(s)	R(#)	Q(#)	T(s)	Perses	Vulcan
clang-22382	144	108	10,363	1,521	105	36,385	2,775	-27.08%	-2.78%
clang-22704	78	62	5,158	2,171	62	13,291	2,490	-20.51%	0
clang-23309	464	303	62,593	9,843	221	245,942	20,266	-52.37%	-27.06%
clang-23353	98	91	7,872	1,389	81	44,377	2,958	-17.35%	-10.99%
clang-25900	239	104	10,382	2,605	102	50,896	4,844	-57.32%	-1.92%
clang-26760	116	56	7,204	4,126	50	38,063	5,471	-56.90%	-10.71%
clang-27137	180	88	14,085	18,593	86	51,767	23,160	-52.22%	-2.27%
clang-27747	117	77	8,480	2,477	73	30,163	3,925	-37.61%	-5.19%
clang-31259	406	276	56,473	17,364	269	216,614	41,935	-33.74%	-2.54%
gcc-59903	316	197	26,095	8,330	194	95,733	12,972	-38.61%	-1.52%
gcc-60116	443	247	59,359	9,768	204	361,654	26,424	-53.95%	-17.41%
gcc-61383	272	209	31,705	13,900	209	86,581	19,025	-23.16%	0
gcc-61917	150	103	10,028	2,963	101	41,453	4,631	-32.67%	-1.94%
gcc-64990	240	204	14,823	5,061	204	44,490	6,781	-15.00%	0
gcc-65383	153	84	4,918	2,150	84	16,532	2,907	-45.10%	0
gcc-66186	328	227	26,615	18,767	188	90,776	32,040	-42.68%	-17.18%
gcc-66375	440	227	30,792	14,918	227	124,161	33,014	-48.41%	0
gcc-70127	301	230	30,502	17,309	223	108,461	29,712	-25.91%	-3.04%
gcc-70586	159	106	18,326	22,434	92	56,063	48,043	-42.14%	-13.21%
gcc-71626	51	38	1,763	104	38	6,012	201	-25.49%	0
median	210	107	14,454	6,696	104	51,332	9,877	-38.11%	-2.40%
mean	235	152	21,878	8,790	141	88,071	16,183	-37.41%	-5.89%

Chapter 4

T-Rec: Fine-Grained Language-Agnostic Program Reduction Guided by Lexical Syntax

This chapter introduces T-Rec, a fine-grained, lexical syntax-guided reduction technique to strengthen canonicalization—a key feature for facilitating bug deduplication. By operating within tokens rather than treating them as atomic, T-Rec reduces superficial differences among programs that essentially trigger the same bug. When integrated into Vulcan, T-Rec successfully eliminates over a thousand additional duplicates and further shrinks reduced programs, demonstrating its substantial impact on both canonicalization and minimization.

4.1 Introduction

For effective and efficient reduction, various language-agnostic program reduction techniques have been proposed in recent decades [226, 152, 192, 170, 110, 114, 161, 221, 200]. Although these techniques have been demonstrated to be effective, their performance, especially canonicalization, is still limited. One of the major bottlenecks is related to their coarse reduction granularity. Specifically, in previous techniques such as HDD and Perses, bug-triggering programs are converted into parse trees (*i.e.*, tree-based), and the subsequent reduction transformations (*e.g.*, deletion) regard each leaf node (*i.e.*, each token) as an atomic element. Therefore, each token (such as an identifier and a string literal) can only either be entirely removed or kept as it is. In other words, these algorithms fail

to recognize the benefits of shrinking and canonicalizing tokens and consequently do not make attempts to incorporate such optimizations. A recently proposed language-agnostic reducer, DDSET, tackles this problem by converting concrete structures in the program to abstract symbols [83]. For example, it can reduce a concrete string token to the abstract string symbol in the corresponding formal grammar, meaning that the content of the string does not affect whether the program triggers the bug or not. However, our evaluation indicates that DDSET is not effective with bug-triggering C programs. One possible explanation is that when the inputs have strict restrictions imposed upon on each token by the semantics or the property (*i.e.*, triggering a certain bug), the abstracting process may become infeasible, and thus the effectiveness of DDSET may significantly deteriorate. C-Reduce [170] is another reducer that attempts to overcome the limitation. It is a widely recognized and powerful reducer commonly employed in compiler testing work and programming language communities [139, 117, 218, 175, 135, 179, 70, 140]. C-Reduce strives to reduce identifiers, strings and numbers to their canonical forms (More details in § 4.3.1). However, the approach used by C-Reduce is specifically optimized for reducing C/C++ programs. Although some parts of its approach are language-agnostic, their performance tends to be limited when the language is less similar to C/C++, and thus could benefit from further enhancements.

Practical Benefits of Fine-Grained Reduction. Fine-grained reduction improves program reduction in two aspects. First, it can minimize tokens such as identifiers and literals to canonical tokens (formally defined in § 4.4.1), thus making the bug-triggering program concise and further increasing the degree of canonicalization. Such improvement can benefit bug deduplication, bug triage, and compiler fuzzing taming [45]. The other benefit of fine-grained reduction is that it can potentially create reduction opportunities for removing more tokens and lead to smaller and simpler reduction results.

T-Rec. Motivated by the practical benefits above, we propose T-Rec, a *fine-grained* language-agnostic program reduction technique. It enhances the capabilities of reduction and canonicalization of existing language-agnostic program reducers by incorporating a novel, fine-grained reduction phase. Meanwhile, since T-Rec is guided by lexical syntax, it can be readily and effectively applied to a wide range of programming languages. T-Rec reduces tokens in two different ways. First, for each token, T-Rec tries to find the canonical token and replace the original token with the canonical one. Specifically, T-Rec leverages the lexical syntax to generate a list of smaller tokens arranged in ascending shortlex order (*i.e.*, shorter tokens come before longer tokens, and tokens with the same length are ordered lexicographically). Then, it searches for the first token in the list that can replace the original one until an attempt limit is reached. The found token is the canonical token and is used to replace the original token. Such a replacement-based approach is especially

effective in scenarios where there is no strict constraint imposed upon the content of the token for the program to trigger the bug. Second, if T-Rec cannot find the canonical token within the attempt limit, T-Rec reduces the token by attempting to remove unnecessary characters in the token with the guidance of lexical syntax and then converts the remaining characters to their canonical forms. More details of this approach are presented in § 4.4. Compared to the replacement-based approach, this method is more effective when some parts of the token are essential for the program to trigger the bug.

We conducted extensive evaluations with two benchmark suites, named **Benchmark-Cano** and **Benchmark-Reduce**. **Benchmark-Cano** was curated by Chen *et al.* [45]. It contains 2,501 test cases that trigger 11 unique crash bugs in GCC 4.3.0, and 1,295 test cases that trigger 35 unique miscompilation bugs also in GCC 4.3.0. We use this benchmark suite for evaluating the performance of T-Rec in facilitating deduplication. **Benchmark-Reduce** is a multi-lingual benchmark suite. It contains 20 C, 20 Rust, and 205 SMT-LIBv2 programs in total, and is used for evaluating the effectiveness, efficiency, and generality of T-Rec. We implemented T-Rec on top of Perses [192], Vulcan [221], and C-Reduce [170], and built three prototypes named T-Rec_{Perses}, T-Rec_{Vulcan}, and T-Rec_{C-Reduce}. Both Perses and Vulcan are open-source syntax-guided language-agnostic reducers, and C-Reduce is a widely used reducer specifically optimized for reducing C/C++ programs. As for baselines, we use Perses, Vulcan, C-Reduce, DDSET and some variants built on top of these existing reducers (more details in § 4.5).

The evaluation results show that all the reducers with T-Rec integrated exhibit significantly better capability in canonicalization than the original base reducers. Specifically, T-Rec_{Perses}, T-Rec_{Vulcan}, and T-Rec_{C-Reduce} eliminate 1,294, 1,315, and 336 more duplicates in **Benchmark-Cano** than Perses, Vulcan, and C-Reduce. The evaluation results also show that in terms of byte size, on average, the results of T-Rec_{Perses} are 65.52%, 28.34%, and 42.86% smaller than those of Perses, and the results of T-Rec_{Vulcan} are 53.73%, 19.79%, and 16.24% smaller than those of Vulcan on C, Rust, and SMT-LIBv2 benchmarks, respectively. Such results further demonstrate the superiority of T-Rec in canonicalization. Moreover, in terms of the number of remaining tokens in the reduced program, T-Rec_{Perses} significantly outperforms Perses and C-Reduce-cano_{Perses} (a variant of T-Rec_{Perses} that uses the non-C/C++-specific canonicalization transformations rather than T-Rec in the fine-grained reduction phase), which demonstrates that T-Rec can better create reduction opportunities for removing more tokens than C-Reduce-cano. Finally, the results also show that the overhead brought by T-Rec is within a reasonable range. Concretely, T-Rec_{Perses} takes 24.22%, 31.22%, and 56.46% more time than Perses on C, Rust, and SMT-LIBv2 benchmarks, respectively, and T-Rec_{Vulcan} takes 1.59%, 5.04%, and 5.01% more time than

Vulcan on each benchmark.

Contribution. We make the following contributions.

- We propose the first work that uses lexical syntax to efficiently and effectively perform fine-grained language-agnostic program reduction at the token level. Specifically, we propose a two-stage canonicalization approach to reduce each token to its canonical form.
- We demonstrate that the fine-grained reduction process can not only significantly increase the degree of canonicalization for a reducer, but also create more reduction opportunities to further remove tokens, thus enhancing the reduction ability of prior reducers.
- Our extensive experiments with a benchmark suite including programs in C, Rust, and SMT-LIBv2 strongly demonstrate the efficacy and efficiency of T-Rec.

4.2 Motivating Example

In the section, we use the benchmarks rust-77002, gcc-60116, and gcc-71626 as motivating examples to illustrate how T-Rec enhances the capabilities of both canonicalization and reduction.

rust-77002. Figure 4.1a, 4.1b, and 4.1c show the reduced programs output by Perses, C-Reduce-cano_{Perses}, and T-Rec_{Perses} on the benchmark rust-77002 respectively. In rust-77002, the bug-triggering program contains an integer token `1_000_000_000_000`. T-Rec successfully reduces it to `0`, while C-Reduce cannot. The reason is that C-Reduce uses regular expression substitutions to canonicalize integers, and the regular expression does not cover integers split with underscores. In contrast, T-Rec can handle any type of integer in any language as it is guided by the lexical syntax. Reducing the original integer token to `0` further narrows down the space of possible root causes of the bug, *i.e.*, it shows that the extremely large value of the variable (`1_000_000_000_000`) is not essential to trigger the bug. Additionally, in this case, any program with a different integer is a duplicate (*i.e.*, a different program that triggers the same bug). With T-Rec, the difference among such duplicates can be completely eliminated, which makes deduplication trivial.

gcc-60116. Figure 4.1d, 4.1e, and 4.1f show the results on the benchmark gcc-60116, which illustrates another two cases where T-Rec performs better than C-Reduce in terms of canonicalization. First, it can be observed that T-Rec reduces the string `"checksum = %X\n"` to `"%X"`, but C-Reduce does not reduce it at all. This is because C-Reduce attempts to replace the string only with an empty string, whereas T-Rec further performs

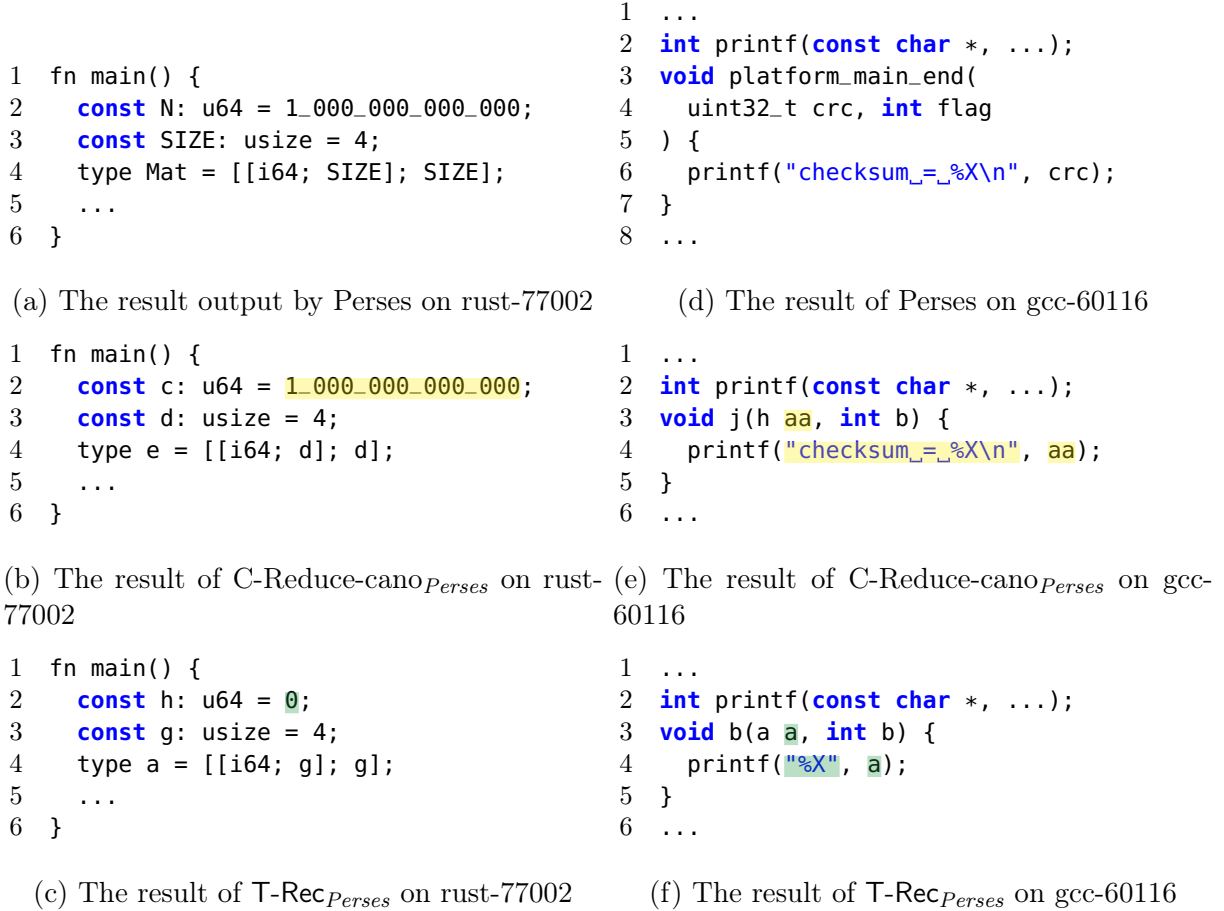


Figure 4.1: Code snippets in programs output by Perses, C-Reduce-cano_{Perses}, and T-Rec_{Perses} on benchmark rust-77002 (left) and gcc-60116 (right).

deletion-based canonicalization if such a replacement fails. Additionally, in this benchmark, T-Rec better canonicalizes identifiers than C-Reduce. The original identifier `crc` is reduced to `a` by T-Rec, whereas C-Reduce reduces it to `aa`. Although there is only a one-character difference in this case, such an improvement can be important in scenarios like bug deduplication. This difference stems from C-Reduce grouping identifier tokens that have the same name and renaming different groups of identifier tokens with different canonical identifiers. Therefore, the number of unique identifiers does not change after canonicalization. In this case, C-Reduce exhausts all the one-letter identifiers. In contrast, T-Rec canonicalizes each single token solely, which can potentially decrease the number of unique identifiers.

gcc-71626. Figure 4.2 shows the results on the benchmark gcc-71626, which illustrates that T-Rec is effective in removing tokens. Figure 4.2a is the reduced program output by Perses, and Figure 4.2b is the result of T-Rec_{Perses}. Compared to Perses, T-Rec_{Perses} further removed lines 2 and 8 (highlighted in red). The reason that Perses cannot remove these two lines is that line 9 in Figure 4.2a uses the variable defined in line 8, and line 8 uses the function defined in line 2. Nevertheless, after the fine-grained reduction performed by T-Rec, the identifier in the curly bracket on line 9, becomes the same as the function name on line 7 in Figure 4.2a (as shown in Figure 4.2b, on line 6 and line 7, the function name and the identifier in the curly bracket are both `b`). This change makes line 2 and 8 in Figure 4.2a obsolete, and thus they can be removed.

1	<code>typedef long llong;</code>	1	<code>typedef long a;</code>
2	<code>test1char8() {}</code>	2	<code>typedef a c</code>
3	<code>typedef llong vllong1</code>	3	<code>--attribute--(</code>
4	<code>--attribute--(</code>	4	<code>--vector_size--(sizeof(a)))</code>
5	<code>(--vector_size--(sizeof(llong)))</code>	5	<code>);</code>
6	<code>);</code>	6	<code>c b() {</code>
7	<code>vllong1 test2llong1() {</code>	7	<code>c a = {b};</code>
8	<code>llong c = test1char8;</code>	8	<code>return a;</code>
9	<code>vllong1 v = {c};</code>	9	<code>}</code>
10	<code>return v;</code>	10	<code>main() {}</code>
11	<code>}</code>		
12	<code>main() {}</code>		

(a) The result of Perses on gcc-71626

(b) The result of T-Rec_{Perses} on gcc-71626

Figure 4.2: Results output by Perses and T-Rec_{Perses} on benchmark gcc-71626.

4.3 Preliminaries

4.3.1 Canonicalization in C-Reduce

C-Reduce has a few transformations specifically implemented for canonicalization. These transformations can be classified into three types by their purposes, *i.e.*, renaming identifiers, deleting content of string literals, and reducing integers. All the renaming transformations except one of them are language-specific as they need to be performed on top of the AST. The one exception does not rely on the AST; instead, it refactors the program by renaming all the identifiers with a set of canonical identifiers (*i.e.*, **a**, **b**, **c**, $\dots$, **aa**, **ab**, **ac**, $\dots$). The other two types of transformations are also not C/C++-specific. The transformation for deleting the content of strings simply replaces each string token with an empty string, and the transformations for reducing integers are based on regular expression substitution (*e.g.*, matching a hex and replacing with a decimal). For convenience, we refer to the group of non-C/C++-specific canonicalization transformations mentioned above as **C-Reduce-cano**. However, the generality of C-Reduce-cano is limited even though they are not C/C++-specific. First, the effectiveness of renaming and string-deleting transformation relies on the correctness of the lexing. However, C-Reduce only utilizes a C lexer, which cannot ensure correctness when being applied to different languages. For example, in SMT-LIBv2, identifiers can include dashes (*i.e.*, -), and C-Reduce cannot recognize such identifiers and thus cannot rename them. Second, the regular expression substitution does not work if the target language supports different forms of integer literals. For example, C-Reduce cannot reduce `1_000_000_000_000` to `0` in rust-77002 shown in Figure 4.1b. In addition to generality, C-Reduce-cano is also limited in terms of effectiveness. First, C-Reduce-cano cannot effectively canonicalize tokens when there are restrictions imposed upon the tokens. For example, C-Reduce-cano cannot reduce the string at all in gcc-60116 shown in Figure 4.1e. Second, C-Reduce-cano can hardly create reduction opportunities for removing more tokens, *e.g.*, they cannot help remove more tokens as T-Rec does in gcc-71626 shown in Figure 4.2.

4.3.2 Lexical Syntax

Lexical syntax determines the way that a sequence of characters should be split into a sequence of tokens. It is described by a list of lexical rules. Each rule defines the form of a specific type of token. Figure 4.3 shows a (partial) lexical rule in the syntax of the C programming language written in ANTLR [7] grammar format, which formally defines the **Constant** token type. The fragment rules (rules that start with the keyword “fragment”)

```

1  Constant
2    : IntegerConstant
3    | FloatingConstant
4    | CharacterConstant ;

6  fragment IntegerConstant
7    : DecConstant IntSuffix?
8    | OctConstant IntSuffix?
9    | HexConstant IntSuffix?
10   | BinaryConstant ;

12 fragment BinaryConstant
13   : '0' [bB] [0-1]+ ;

15 fragment DecConstant
16   : NonzeroDigit Digit* ;

18 fragment OctConstant
19   : '0' OctalDigit* ;

21 fragment HexConstant
22   : HexPrefix HexDigit+ ;

24 fragment HexPrefix
25   : '0' [xX] ;

27 fragment NonzeroDigit
28   : [1-9] ;

29 fragment OctalDigit
30   : [0-7] ;

32 fragment HexDigit
33   : [0-9a-fA-F] ;

35 fragment IntSuffix
36   : USuffix LongSuffix?
37   | USuffix LLSuffix
38   | LongSuffix USuffix?
39   | LLSuffix USuffix? ;

41 fragment USuffix
42   : [uU] ;

44 fragment LongSuffix
45   : [lL] ;

47 fragment LLSuffix
48   : 'll'
49   | 'LL' ;

51 fragment Digit
52   : [0-9] ;

```

Figure 4.3: The partial lexical grammar of **Constant** token in C written in ANTLR grammar format

are introduced for better readability. This example rule defines that a **Constant** can be an **IntegerConstant**, a **FloatingConstant**, or a **CharacterConstant** (line 1-4). Then, **IntegerConstant** is defined to have four different forms (line 6-10). For example, it can be a **HexConstant** optionally (indicated by **?**) followed by an **IntSuffix** (line 9). **HexConstant**, according to the corresponding fragment rule, is defined to be a **HexPrefix** followed by one or multiple (indicated by **+**) **HexDigit** (line 21-22). A **HexPrefix** can be either **0x** or **0X**, indicated by '**0**' [**xX**] (line 24-25), while a **HexDigit** can be any character in the set [**0-9a-fA-F**] (line 32-33). Guided by the lexical rule, we can possibly split a token into finer structures. For example, a hexadecimal constant token **0xff00ull**, according to the example lexical rule, can be divided into four components:

Component	Fragment	Description
0x	HexPrefix	the prefix for hexadecimal numbers
ff00	HexDigit+	hexadecimal digits
u	USuffix	the suffix for unsigned numbers
ll	LLSuffix	the suffix for the integer type long long

4.4 Methodology

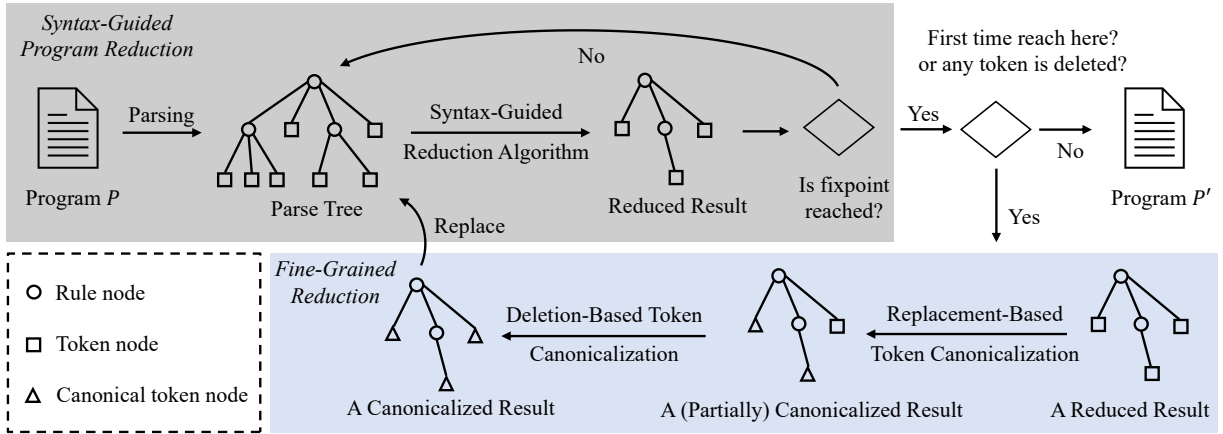


Figure 4.4: The overview of T-Rec.

Figure 4.4 shows the overview of T-Rec. Given a program to reduce, T-Rec first reduces it with a syntax-guided program reduction algorithm. Once the algorithm reaches a fixed point (*i.e.*, it can no longer remove more tokens), the process of **fine-grained reduction**

starts. In this process, T-Rec strives to reduce each token to its canonical form. Specifically, T-Rec first attempts to replace each token with the canonical token, which is generated based on the lexical rule of that particular token type. Such a replacement-based approach is the most efficient way to canonicalize a token. It is also effective in most scenarios, and thus it is performed first. However, some of the tokens may not be able to be directly replaced with the canonical tokens. There might be some constraints imposed upon the tokens to ensure that the program preserves the property (*e.g.*, an integer has to be larger than a certain value). For such tokens, T-Rec further performs a deletion-based approach to canonicalize them. This approach first converts the token to a tree representation that reflects the lexical structure of the token (referred to as a *lex tree*) based on the lexical syntax. Next, it applies lexical syntax-guided reduction to the lex tree to remove unnecessary token fragments. Finally, T-Rec canonicalizes each remaining token fragment by replacing it with the canonical one. This second approach is not as efficient as the first one. Nevertheless, it is more effective when there are strict restrictions imposed upon the token.

Algorithm 5 is the main algorithm of T-Rec. The invocations of `reduceUntilFixpoint` on line 1 and line 13, correspond to the repeating syntax-guided reduction algorithm in Figure 4.4. The for loop from line 5 to line 12 corresponds to the fine-grained reduction process, which will be introduced in detail in § 4.4.2 and § 4.4.3. The `while` condition on line 14 ensures that the reduction terminates when the outer fixpoint is reached.

4.4.1 Definition of Canonical Tokens

Before introducing the two canonicalization approaches, we first define canonical tokens in this subsection.

Definition 4.4.1 (Token). *A token t consists of a list of characters $[c_1, c_2, \dots, c_n]$. It is associated with a type defined by the lexical syntax, denoted as $\text{type}(t)$, and its length equals the number of characters in the list (i.e., n), denoted as $\text{length}(t)$.*

Example. In C programs, `0x00ff` is typically defined as a `Constant` token. Its length is 6 as it contains 6 characters.

Definition 4.4.2 (Shortlex Order). *Given two tokens $t_1 = [c_1^1, c_2^1, \dots, c_m^1]$ and $t_2 = [c_1^2, c_2^2, \dots, c_n^2]$, $t_1 <_s t_2$ if and only if Equation (4.1) or Equation (4.2) holds:*

$$\text{length}(t_1) < \text{length}(t_2) \tag{4.1}$$

$$\text{length}(t_1) = \text{length}(t_2) \wedge \exists i, \forall j < i \implies c_j^1 = c_j^2 \wedge c_i^1 < c_i^2 \tag{4.2}$$

Algorithm 5: The Main Algorithm – $\text{reduce}(P, \psi, G)$

Input : P , the current program to be further reduced
Input : $\psi: \mathbb{P} \rightarrow \mathbb{B}$, the property checking function
Input : G : the formal syntax of the programming language
Output : A reduced program that still satisfies ψ
// syntax-guided program reduction

```
1  $P_{min} \leftarrow \text{reduceUntilFixpoint}(P, \psi)$ 
2 do// Token canonicalization loop
3    $P_{prev} \leftarrow P_{min}$ 
4    $\text{tokenCount} \leftarrow P_{min}.\text{tokenCount}()$ 
5   for  $i \in [1, \text{tokenCount}]$  do
6      $\text{origToken} \leftarrow P_{min}.\text{getNthToken}(i)$ 
7      $P'_{min} \leftarrow \text{replacementBasedCanonicalize}(P_{min}, \text{origToken}, \psi, G)$ 
8     if  $P'_{min} \neq P_{min}$  then
9        $P_{min} \leftarrow P'_{min}$ 
10    else
11      // deletion-based token canonicalization
12       $P_{min} \leftarrow \text{shrinkToken}(P_{min}, \text{origToken}, \psi, G)$ 
13       $P_{min} \leftarrow \text{canonicalizeTokenFragments}(P_{min}, \text{origToken}, \psi, G)$ 
14  $P_{min} \leftarrow \text{reduceUntilFixpoint}(P_{min}, \psi)$ 
15 while  $P_{min}.\text{size}() < P_{prev}.\text{size}()$ 
16 return  $P_{min}$ 
```

Example. Given three string literal tokens: t_1 is "", t_2 is "a", and t_3 is "b", it can be deduced that $t_1 <_s t_2 <_s t_3$ because t_1 contains the smallest number of characters (*i.e.*, two double quotes), and the character **a** in t_2 is in front of the **b** in t_3 in lexicographic order.

Definition 4.4.3 (Canonical Token). *Given a program P that consists of a list of tokens $[t_1, t_2, \dots, t_n]$ and a property ψ that P has (*i.e.*, $\psi(P) = \text{true}$), a token t_i is canonical if and only if:*

$$\forall t'_i <_s t_i \wedge \text{type}(t'_i) = \text{type}(t_i), P' = P[t_i/t'_i] \implies \psi(P') = \text{false} \quad (4.3)$$

where $\text{type}(t)$ is the token type of the token t defined by the lexical syntax (*e.g.*, identifier), and $P[t_i/t'_i]$ the program derived by replacing t_i with t'_i in P .

Example. Suppose that a program P that contains four tokens, `[print, (, "hello", )]`, can print "hello", and assume that the property ψ is that the program prints at least one ASCII printable character (character code 32-127). Then, the canonical token of the string token "hello" is " " (*i.e.*, a string that only contains a space), because " " is the first string token in the shortlex order that preserves the property of the program. Sometimes, following the shortlex order based on ASCII can lead to canonical tokens with poor readability. To fix this limitation, a feasible solution is to customize the order of the alphabet rather than following the conventional order in ASCII.

4.4.2 Replacement-Based Token Canonicalization

Algorithm 6 describes the way that T-Rec replaces tokens with generated canonical tokens. The algorithm takes four inputs, *i.e.*, the current minimal program, the token to be replaced, the property checking function, and the formal syntax of the corresponding programming language. First, T-Rec retrieves the lexical rule of the token from the given formal syntax based on the token type (line 1). Next, with the obtained lexical rule, it generates a list of tokens in shortlex order (line 2). The generation approach that we use is similar to the input generation method mentioned by a few existing grammar fuzzing studies [13, 185, 99], but instead of random generation, we make the generation follow the shortlex order. Typically, the result of the generation is determined by a list of choices that decide (1) whether to generate for a part marked with an optional quantifier, (2) how many times to generate a part marked with Kleene star and Kleene plus, (3) and which character(s) within the defined set to generate. The strategy that T-Rec follows is generating as few parts as possible and always prioritizing character(s) that are sequentially positioned in the front in shortlex order. According to the definition of canonical tokens, the first token in `tokenList` that makes the program pass the property test is the canonical token. Therefore, in the following loop (line 4-7), T-Rec iterates over `tokenList` until it finds

Algorithm 6: replacementBasedCanonicalize(P , origToken, ψ , G)

Input : P , the current best program
Input : origToken, the token to be replaced
Input : ψ , the property checking function, which takes as input a program and outputs a boolean
Input : G : the formal syntax of the programming language
Output : A reduced program that has the property ψ

```
1 lexicalRule  $\leftarrow$  getLexicalRule(origToken.tokenType,  $G$ )
2 tokenList  $\leftarrow$  generateInShortlexOrder(lexicalRule)
3 attemptLimit  $\leftarrow$  getAttemptLimit(origToken.tokenType)
4 for  $i \in [1, \min(\text{tokenList.size}(), \text{attemptLimit})]$  do
5   newToken  $\leftarrow$  tokenList[ $i$ ]
6   if replaceIfPassTest( $P$ ,  $\psi$ , origToken, newToken) then
7     break
8 return  $P$ 
9 Function replaceIfPassTest( $P$ ,  $\psi$ , origToken, newToken):
10   // try to replace tokens with the same text simultaneously
11    $P' \leftarrow$  a new program derived by replacing all the tokens in  $P$  that have the same text
12   as origToken with newToken
13   if  $\psi(P')$  then
14     update  $P$  with  $P'$ 
15     return true
16   // if not work, try to replace a single token
17    $P' \leftarrow$  a new program derived by replacing origToken in  $P$  with newToken
18   if  $\psi(P')$  then
19     update  $P$  with  $P'$ 
20     return true
21   return false
```

the token that can replace the original token while preserving the property ψ (line 6-7).

Sometimes, the canonical token is not positioned in the front in the shortlex-ordered token list. For example, it is possible that to preserve the property, an integer literal needs to be larger than or equal to 100. In such a case, T-Rec has to try all the integers from 0 to 99 to eventually find that 100 is the canonical token. To ensure the replacement-based token canonicalization algorithm does not waste too much time for such cases, T-Rec terminates the for loop once the attempt limit set for the token type is reached (line 3 and line 4). T-Rec has two different attempt limits for different token types. For identifiers, the attempt limit is set to infinite. Because for identifiers, the aforementioned problem does not exist as the text content of the identifier does not affect the semantics of the program. For tokens other than identifiers, the attempt limit is set to a small value. In our implementation, it is set to 2 in order to cover both 0 and 1 for integer tokens.

Property Checking with New Tokens. The function `replaceIfPassTest` in Algorithm 6 describes the way that T-Rec checks whether a new token can replace the original token(s) without making the program lose the property. As shown in the algorithm, T-Rec performs replacement in two different ways. It first replaces all the tokens in t that have the same text as `origToken` with `newToken` to generate a new program (line 10), and it performs the property test on this new program (line 11-13). The insight behind this replacement is that tokens having the same text are often semantically related, and thus replacing them together sometimes better preserves the property of the program. If the new program still has the property (*i.e.*, $\psi(P) = \text{true}$), T-Rec updates the current minimal program P and returns `true` (line 12-13). Otherwise, T-Rec generates another new program by only replacing the single token `origToken` in P with `newToken` (line 14), and T-Rec repeats the previous steps, *i.e.*, testing and possibly updating, with this new program (line 15-17). If neither of these two new programs passes the property, the function returns `false`, indicating the new token fails the test.

Example. Figure 4.5 illustrates the process of a new token being tested by the function `replaceIfPassTest`. Assuming that the original program is P_1 in Figure 4.5a, the original token is `long_var` on line 6, and the new token is `a`. The function `replaceIfPassTest` first generates P_2 as shown in Figure 4.5b by replacing all the `long_var` in the original program P_1 with `a`. If P_2 passes the property test ψ , the function updates P with P_2 and returns `true` to indicate that the test is passed. Otherwise, if P_2 does not pass the property test ψ , the function `replaceIfPassTest` generates P_3 in Figure 4.5c by replacing only the `long_var` on line 6, since it is the original token. Next, the function `replaceIfPassTest` tests P_3 against the property test ψ . If the property test is passed, the program P is updated and `true` is returned; otherwise, the function returns `false`, indicating the test of this new token fails.

<pre> 1 // The original program 2 // 'long_var' on line 6 3 // is the token 4 // to be replaced 5 int main() { 6 int long_var = 1; 7 int b = long_var + 2; 8 return b; 9 }</pre>	<pre> 1 // First program to test 2 // 3 // All 'long_var's 4 // are replaced 5 int main() { 6 int a = 1; 7 int b = a + 2; 8 return b; 9 }</pre>	<pre> 1 // Second program to test 2 // 3 // Only 'long_var' on 4 // line 6 is replaced 5 int main() { 6 int a = 1; 7 int b = long_var + 2; 8 return b; 9 }</pre>
(a) The original program	(b) The generated program P_2	(c) The generated program P_3

Figure 4.5: An example of property checking with a new token

4.4.3 Deletion-Based Token Canonicalization

Replacement-based token canonicalization is not efficient when the token to be canonicalized is imposed with certain restrictions for preserving the property of the program (*e.g.*, an integer needs to be at least 100 to make the program have the property). Therefore, we propose a deletion-based token canonicalization approach to canonicalize the remaining tokens. The intuition of this approach is to first remove the unnecessary characters or token fragments (*i.e.*, `shrinkToken`) and then canonicalize the remaining token fragments (*i.e.*, `canonicalizeTokenFragments`).

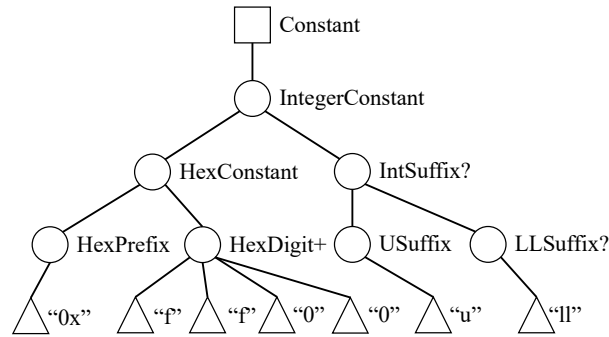


Figure 4.6: The lex tree of token `0xff00ull` in C language.

Lex Tree

To facilitate the reduction with lexical syntax, we utilize *lex tree* to represent the internal structures of tokens. *Lex tree* is similar to a parse tree, but instead of representing a program, *lex tree* only represents a single token. Lex trees not only enable T-Rec to perform a tree-based lexical syntax-guided reduction algorithm to effectively and efficiently remove unnecessary parts within a token but also contain the necessary information for canonicalizing the shrunk token.

Figure 4.6 shows an example lex tree which is built from a C constant token `0xff00ull`, based on the lexical definition of `Constant`.¹ The root node of the tree represents the corresponding lexical rule. Each intermediate node represents a fragment, and each leaf node is an atomic fragment. Such a tree representation can provide guidance for token shrinking. First, it can be observed that the first two characters `0` and `x` are bound together. They form the atomic fragment `HexPrefix`, and either deleting them or one of them will probably make the token syntactically invalid. Next, `ff00` are four hexadecimal digits. Their parent node is a *Kleene plus node* (indicated by `+`). For every Kleene plus node, its child can be deleted without breaking the syntax as long as at least one child remains. The remaining parts `u` and `ll` are two suffixes of an integer. The former is an `unsigned` suffix and the latter is a `long long` suffix. Their lowest common ancestor is `IntSuffix`, which is an *optional node* (indicated by `?`). For such a node, the sub-tree rooted in it can be deleted without breaking the syntax. Another special type of node that is not included in the example is *Kleene star node* (indicated by `*`). For Kleene star nodes, deleting an arbitrary number of its children does not break the syntax.

In addition to providing guidance to token shrinking, lex trees also contain the information for canonicalization. For example, according to the lex tree, the four characters in `ff00` are all `HexDigit`. According to the formal syntax, each of them can be a character in `[0-9a-fA-F]`. Therefore, we can canonicalize `ff00` by replacing each token fragment with another valid one that is positioned as close to the beginning as possible in shortlex order, *e.g.*, the resulting four characters can be `1000`.

Lexical Syntax-Guided Reduction

As introduced in § 4.4.3, there are three special types of nodes in a lex tree: optional nodes, Kleene star nodes, and Kleene plus nodes. For nodes that belong to these types,

¹A token definition can simply be a regular expression. In such cases, T-Rec recognizes and outlines each necessary, optional, and repetitive fragment in the regular expression and builds lex trees based on the new outlined definition.

Algorithm 7: Lexical Syntax-Guided Reduction – `shrinkToken( $P$ , origToken,  $\psi$ ,  $G$ )`

Input : P , the current best program
Input : origToken, the token to be shrunk
Input : ψ , the property checking function, which takes as input a program and output a boolean
Input : G : the formal syntax of the programming language
Output : A reduced program that has the property ψ

```

1  $t_{lex} \leftarrow \text{buildLexTree}(\text{origToken}, G)$ 
2  $\text{queue} \leftarrow [t_{lex}.\text{root}]$ 
3 while  $\text{queue}$  is not empty do
4    $\text{node} \leftarrow \text{queue.poll}()$ 
5   if  $\text{node}$  is an optional node then
6      $t'_{lex} \leftarrow t_{lex}.\text{removeNode}(\text{node})$ 
7     if  $\text{replaceIfPassTest}(P, \psi, \text{treeToToken}(t_{lex}), \text{treeToToken}(t'_{lex}))$  then
8        $t_{lex} \leftarrow t'_{lex}$ 
9     else
10       $\text{queue.addAll}(\text{node.getChildren}())$ 
11   else if  $\text{node}$  is a Kleene star node or a Kleene plus node then
12      $\text{children} \leftarrow \text{node.getChildren}()$ 
13      $\text{remaining} \leftarrow \text{ddmin}(\text{children}, \psi)$ 
14     update  $t_{lex}$  by removing nodes in children but not in remaining
15     update  $P$  with the updated  $t_{lex}$ 
16      $\text{queue.addAll}(\text{remaining})$ 
17   else  $\text{queue.addAll}(\text{node.getChildren}())$ 
18 return  $P$ 

```

deleting them or their children in a certain way can ensure that the resulting token is still syntactically valid. Based on this insight, we propose a lexical syntax-guided reduction algorithm.

Algorithm 7 describes the algorithm in detail. It takes three inputs, the current minimal program P (*i.e.*, the fixpoint result), the token to be shrunk `origToken`, and the property test function ψ which checks whether the program still has the property. First, T-Rec builds the lex tree for the given token (line 1) and initializes `queue` with the root node of the lex tree (line 2). Next, T-Rec reduces the lex tree in a **while** loop (line 3-17). Whenever `queue` is not empty, T-Rec enters the loop and retrieves the first node in the queue (line 4). Depending on the type of the node, T-Rec takes the following actions.

- **Optional Node** If this node is an optional node, T-Rec generates a new lex tree by removing this optional node from the original lex tree (line 6). If the new lex tree t'_{lex} passes the property test, the original lex tree t_{lex} is updated with t'_{lex} , and no nodes are added to the queue (line 7-8); otherwise, if t'_{lex} does not pass the test, T-Rec adds all the children of the optional node to the queue (line 9-10), and t'_{lex} is discarded.
- **Kleene Star Node** or a **Kleene Plus Node** If the node is a Kleene star node or Kleene plus node, T-Rec applies `ddmin` to minimize the number of children of the node. Next, t_{lex} is updated by only keeping the remaining children (line 12-14), and the queue is updated by adding all the remaining children to it (line 16).
- Any nodes that do not belong to these three types are not processed, and T-Rec simply adds their children to the queue (line 17).

Lex Tree-Based Canonicalization

After the lexical syntax-guided reduction finishes, T-Rec canonicalizes the token by replacing each token fragment with another valid one that is positioned as close to the beginning as possible in shortlex order as shown in Algorithm 8 (line 3-10). The canonicalization process is similar to replacement-based token canonicalization. Replacement-based token canonicalization canonicalizes the entire program by replacing each token with a canonical one, while lex tree-based canonicalization canonicalizes a single token by replacing each token fragment with a canonical one. Note that there is also an attempt limit in the algorithm (line 6). In our implementation, we set it to 2, which is the same value in replacement-based token canonicalization for tokens other than identifiers.

Example. To illustrate the process of the deletion-based token canonicalization step by step, we use `0xff00ull` as an illustrating example and show how this token is canonicalized.

Algorithm 8: canonicalizeTokenFragments(P , origToken, ψ , G)

Input : P , the current best program
Input : origToken, the token to be canonicalized
Input : ψ , the property checking function, which takes as input a program and output a boolean
Input : G : the formal syntax of the programming language
Output : A canonicalized program that has the property ψ

```

1  $t_{lex} \leftarrow \text{buildLexTree}(\text{origToken}, G)$ 
2  $\text{leaves} \leftarrow t_{lex}.\text{getLeaves}()$ 
3 for  $i \in [1, \text{leaves.size}()]$  do
4    $\text{fragment} \leftarrow \text{leaves}(i)$ 
5    $\text{fragmentList} \leftarrow \text{generateFragmentsInShortlexOrder}(t_{lex}, \text{fragment}, G)$ 
6   for  $j$  in  $\text{range}(\min(\text{fragmentList.size}(), \text{attemptLimit}))$  do
7      $\text{newFragment} \leftarrow \text{fragmentList}[j]$ 
8      $t'_{lex} \leftarrow$  a new lex tree derived by replacing fragment with newFragment
9     if  $\text{replaceIfPassTest}(P, \psi, \text{treeToToken}(t_{lex}), \text{treeToToken}(t'_{lex}))$  then
10      break
11 return  $P$ 

```

Original Token	0xff00ull	New Token	ψ	Updated Token
① remove IntSuffix?:	0xff00 <u>ull</u>	→ 0xff00	→ False	→ 0xff00ull
② reduce HexDigit+:	0xff <u>00</u> ull	→ 0xfffull	→ False	→ 0xff00ull
③ reduce HexDigit+:	0x <u>ff</u> 00ull	→ 0x00ull	→ False	→ 0xff00ull
④ reduce HexDigit+:	0xff0 <u>0</u> ull	→ 0xff0ull	→ True	→ 0xff0ull
⑤ remove LLSuffix?:	0xff0u <u>ll</u>	→ 0xff0u	→ True	→ 0xff0u
⑥ replace 1st HexDigit:	0x <u>f</u> f0u	→ 0x0f0u	→ False	→ 0xff0u
⑦ replace 1st HexDigit:	0x <u>f</u> f0u	→ 0x1f0u	→ True	→ 0x1f0u
⑧ replace 2nd HexDigit:	0x1 <u>f</u> 0u	→ 0x100u	→ True	→ 0x100u

Figure 4.7: An example showing the process of the deletion-based token canonicalization.

The process is shown in Figure 4.7, using the lex tree of `0xff00ull` shown in Figure 4.6. The lexical syntax-guided reduction algorithm visits nodes in the breadth-first order, and thus `IntSuffix?` is the first special node being visited. Since it is an optional node, the algorithm removes the entire node, which creates a new token `0xff00` (step ①), and invokes `replaceIfPassTest` to test this new token. Assuming the new token fails the test, the algorithm discards the new token, and continues to visit subsequent nodes. The next special node is `HexDigit+`. For this Kleene plus node, the algorithm invokes `ddmin` to reduce the four children (step ② - ④). Assuming that eventually `ddmin` removes the last `HexDigit` (*i.e.*, 0) (to ensure the result is a fixpoint, more steps are performed by `ddmin` after step ④, we assume all these steps fail the property test and omit them for brevity), the token is then updated with the smaller one, and the program is updated accordingly. The last special node is `LLSuffix?`, which is also an optional node. Similarly, the algorithm generates a new token by removing the node. If this new token passes the test, the algorithm updates the lex tree and the program again. Eventually, the lexical syntax-guided reduction algorithm shrinks the token from `0xff00ull` to `0xff0u`. Next, T-Rec performs the lex tree-based canonicalization algorithm. Since the first token fragment `0x` does not have any other valid alternatives, T-Rec just skips it and moves to the next token fragment, *i.e.*, the first `f`. From the formal syntax, this fragment is defined by `[0-9a-fA-F]`. The first two fragments in this set in shortlex order are `0` and `1`. Therefore, T-Rec first attempts to replace the first `f` with `0` (step ⑥). If such a replacement fails, T-Rec attempts to replace it with `1` (step ⑦). Suppose such a replacement passes the property test, T-Rec then updates the token and the program and moves to the next fragment and repeats the same process (step ⑧).

4.5 Evaluation

To demonstrate the effectiveness and efficiency of T-Rec, we implemented prototypes of T-Rec on top of Perses, Vulcan, and C-Reduce named `T-RecPerses`, `T-RecVulcan`, `T-RecC-Reduce`² respectively and conducted extensive evaluations to investigate the following research questions:

RQ1: What is the effectiveness of T-Rec in facilitating deduplication?

RQ2: What is the effectiveness of T-Rec in reducing byte size?

RQ3: What is the effectiveness of T-Rec in removing tokens?

²`T-RecC-Reduce` is implemented by replacing C-Reduce-cano (*i.e.*, the non-C/C++-specific canonicalization transformations) in C-Reduce with T-Rec

RQ4: How efficient is T-Rec?

Benchmark Suites. Two benchmarks suites are used for evaluating T-Rec. The first benchmark suite is collected and used by Chen *et al.* [45] (referred to as **Benchmark-Cano**). We use this benchmark for evaluating the performance of T-Rec in facilitating deduplication. The other benchmark suite used for evaluating T-Rec is named **Benchmark-Reduce**. This benchmark suite contains bug-triggering programs in three different programming languages, *i.e.*, C, Rust, and SMT-LIBv2. It is included to further assess the effectiveness and efficiency of T-Rec. Meanwhile, this multi-lingual benchmark suite also allows us to evaluate the generality of T-Rec to some extent.

- **Benchmark-Cano** includes 2,501 test cases that trigger 11 unique crash bugs and 1,295 test cases (1,298 in total, but we cannot reproduce the bug with 3 test cases) that trigger 35 unique miscompilation bugs in GCC 4.3.0. All the test cases are generated by CSmith [223] version 2.1.0 under its default setting. This benchmark suite also contains JavaScript bug-triggering test cases. We did not use them since we did not have access to the corresponding property test scripts.
- **Benchmark-Reduce** consists of 20 C programs, 20 Rust programs, and 205 SMT-LIBv2 programs. The C programs are also used for evaluation in previous studies [192, 221, 209, 230, 78, 228], which is generated by fuzzing techniques including CSmith and EMI [117]. The Rust benchmarks are all real-world bugs collected by the authors, and most of them are also used for evaluation in previous research studies [221, 230, 228]. Each C and Rust benchmark consists of a bug-triggering program that triggers an unique crash or miscompilation bug and a shell script that checks whether a program triggers the bug in certain affected version(s) of compilers. The SMT-LIBv2 programs are collected from the benchmark suite for evaluating ddSMT [114, 161], which contains 241 SMT-LIBv2 programs in total. We used 205 of them in our evaluation since the remaining 36 SMT-LIBv2 programs cannot be parsed or pass the property check.

Baselines. The baselines to which we compared T-Rec include Perses, Vulcan, C-Reduce, DDSET, and a few variants built based on these reducers.

- **Perses** is a language-agnostic syntax-guided program reducer [192]. It excels at efficiency, and reducers including Vulcan and DDSET are built on top of Perses.
- **Vulcan** is the state-of-the-art language-agnostic program reducer [221]. It integrates three auxiliary reducers on top of Perses to trade off execution time for smaller outcomes.
- **DDSET** is another language-agnostic reducer built on top of Perses [83]. It strives to abstract the bug-triggering input after reduction, which can be considered as a differ-

ent approach of canonicalization. We use the implementation from the DDSET GitHub repository [84] and follow the instructions provided in the repository.

- **C-Reduce** is a commonly used state-of-the-art program reducer that is specifically optimized for reducing C/C++ programs. For C-Reduce, in addition to its default setting (referred to as C-Reduce), we also include C-Reduce with its `sllooww` flag enabled (C-Reduce under this setting is referred to as **C-Reduce-slow**) as a baseline. When the flag `sllooww` is enabled, C-Reduce performs more exhaustive reduction transformations and thus provides enhanced capability of both reduction and canonicalization at the cost of execution time.
- **C-Reduce-cano_{Perses}** and **C-Reduce-cano_{Vulcan}** are two variants of T-Rec_{Perses} and T-Rec_{Vulcan}. They are built by replacing the find-grained reduction components (blue part in Figure 4.4) with C-Reduce-cano.

The name convention we followed for naming the prototypes and variants (reducer with a subscript in its name) is that the base name refers to the canonicalization approach, and the subscript refers to the main reduction algorithm.

Metrics. In the evaluation, the following four metrics are used to measure the performance of program reduction:

- **Number of Eliminated Programs:** the total number of eliminated programs. Given a list of programs, after reduction, some programs may become textually equivalent. For each group of textually equivalent programs, only one of them is kept and others are eliminated. A higher value of this metric indicates a better capability of the corresponding reducer in canonicalization.
- **Program Size in Bytes:** the number of in bytes it takes to store the reduced program. This metric can reflect the capability of both reduction and canonicalization to some extent. The smaller the program size, the better the performance of both the reduction and canonicalization.
- **Token Count:** the number of tokens in the reduced program. This metric mainly measures the capability of reduction. The smaller the number of tokens, the better the performance of the reduction.
- **Time:** the execution time (in seconds) of the reduction.

We measure both the program size in bytes and token count of the results to provide a comprehensive evaluation of the effectiveness. The number of tokens reflects the reduction

capability of a reducer. Fewer tokens in the output program indicate better reduction capability. While the program size in bytes also reflects the reduction capability, it reveals the canonicalization capability of a reducer to some extent. Having fewer redundant bytes in the reduced program often leads to fewer differences among reduced duplicates. To observe the reduction performance gap between T-Rec and the baselines in a straightforward way, we also include **percentage change in size** (for both program size in bytes and token count) in the evaluation results, which is calculated by $(\text{size} - \text{size}_{\text{baseline}}) / \text{size}_{\text{baseline}}$.

Different tools may output reduced programs in different formats, and the byte size of the programs can be noticeably affected by their format. To eliminate the influence of format on size comparison, we do not count any whitespace in the reduced programs.

Experiment Setup. The experiments are run on an Ubuntu 20.04 server with AMD Ryzen 9 7950X 16-Core CPU and 128 GB RAM. All the experiments are run with a single thread.

Table 4.1: The reduction results produced by different reducers on the 2,501 crash test cases in **Benchmark-Cano**. The column **Eliminated (#)** shows the total number of programs eliminated by each reducer.

Reducer	Eliminated (#)	Average Time (s)	Average Size (# of Tokens)
Perses	3	83.05	59.87
C-Reduce-cano _{Perses}	741	84.52	59.87
T-Rec _{Perses}	1,297	89.76	55.73
Vulcan	13	146.97	42.27
C-Reduce-cano _{Vulcan}	808	148.05	42.27
T-Rec _{Vulcan}	1,327	157.71	40.80
C-Reduce	1,143	287.09	35.29
T-Rec _{C-Reduce}	1,466	375.48	33.90
C-Reduce-slow	1,959	768.67	31.73
T-Rec _{C-Reduce-slow}	2,053	778.68	30.34
DDSET	3	805.33	N/A

Table 4.2: The reduction results produced by different reducers on the 1,295 miscompilation test cases in **Benchmark-Cano**. The Eliminated (#) column shows the total number of programs that are eliminated because they are textually equivalent to other programs.

Reducer	Eliminated (#)	Average Time (s)	Average Size (# of Tokens)
Perses	0	394.36	285.03
C-Reduce-cano _{Perses}	0	430.09	284.42
T-Rec _{Perses}	0	511.41	228.59
Vulcan	0	2,134.71	197.72
C-Reduce-cano _{Vulcan}	0	2,146.40	197.17
T-Rec _{Vulcan}	1	2,223.95	188.88
C-Reduce	36	1,247.19	79.26
T-Rec _{C-Reduce}	49	1,304.69	73.43
C-Reduce-slow	129	3,468.54	74.79
T-Rec _{C-Reduce-slow}	163	3,511.17	69.14
DDSET	0	5,203.80	N/A

4.5.1 RQ1: Effectiveness in Deduplication

A highly canonicalizing reducer tends to reduce different test cases that trigger the same bug to similar or even identical test cases. Therefore, one meaningful perspective to evaluate the capability of a reducer in canonicalization is to evaluate whether it can reduce duplicates (*i.e.*, test cases that trigger the same bug) to a textually identical test case. In this experiment, we compare the performance of different program reducers in deduplication using **Benchmark-Cano**. Specifically, we utilize each reducer to reduce all the test cases in the benchmark suite. Among the reduced test cases, if there are textually equivalent test cases, only one of them is kept and the others are eliminated. We measure the number of eliminated test cases, the average time to reduce, and the average size of the reduced test cases for each reducer. Table 4.1 and Table 4.2 show the results of the experiment.

Perses vs. C-Reduce-cano_{Perses} vs. T-Rec_{Perses}. T-Rec_{Perses} is a prototype of T-Rec implemented on top of Perses, and C-Reduce-cano_{Perses} integrates C-Reduce-cano instead of T-Rec on top of Perses to serve as the fine-grained reduction technique. As shown in Table 4.1, C-Reduce-cano_{Perses} and T-Rec_{Perses} eliminate 741 and 1,297 textually equivalent programs after reducing the 2,501 crash test cases, whereas Perses eliminates only 3 pro-

grams. With respect to the 1,295 miscompilation test cases, none of these three reducers successfully produces textually equivalent programs and thus no program is eliminated. Such a result indicates that both C-Reduce-cano and T-Rec can effectively canonicalize programs to some extent, and T-Rec is even more effective. Additionally, it should be noted that the results of T-Rec_{Perses} contain fewer tokens than Perses. Specifically, on average, the results of T-Rec_{Perses} have 5.01% and 18.52% fewer tokens than those of Perses on crash and miscompilation test cases, respectively. Timewise, C-Reduce-cano_{Perses} and T-Rec_{Perses} spend 2.91% and 14.79% more time than Perses on the crash test cases and 10.18% and 29.62% more time on the miscompilation test cases.

Vulcan vs. C-Reduce-cano_{Vulcan} vs. T-Rec_{Vulcan}. Similar to C-Reduce-cano_{Perses} and T-Rec_{Perses}, C-Reduce-cano_{Vulcan} and T-Rec_{Vulcan} integrate C-Reduce-cano and T-Rec on top of Vulcan, respectively. The comparison among these three reducers further supports the conclusion that both C-Reduce-cano and T-Rec can effectively canonicalize programs to some extent, and T-Rec is more effective than C-Reduce-cano. Specifically, T-Rec_{Vulcan} can eliminate 519 ($= 1327 - 808$) more crash test cases than C-Reduce-cano_{Vulcan}, and among the three reducers, T-Rec_{Vulcan} is the only one that can eliminate a miscompilation test case. Timewise, T-Rec_{Vulcan} takes 8.49% and 5.77% more time than Vulcan on crash and miscompilation test cases, respectively.

C-Reduce and C-Reduce-slow vs. T-Rec_{C-Reduce} and T-Rec_{C-Reduce-slow}. To further evaluate the capability of T-Rec in deduplication, we implement T-Rec_{C-Reduce} by replacing C-Reduce-cano with T-Rec, and compare this prototype to C-Reduce with the default setting and with the `sllooww` flag enabled. As shown in Table 4.1 and Table 4.2, the T-Rec technique helps C-Reduce and C-Reduce-slow further eliminate 323 and 94 crash test cases as well as 13 and 34 miscompilation test cases. Moreover, on average, the results of T-Rec_{C-Reduce} contain 2.84% and 6.04% fewer tokens than those of C-Reduce on crash test cases and miscompilation test cases, respectively. Correspondingly, the percentage decrease in token size of T-Rec_{C-Reduce-slow} compared to C-Reduce-slow is 2.84% and 4.20%, respectively. In terms of execution time, T-Rec_{C-Reduce} spends 37.48% and 5.12% more time than C-Reduce, and T-Rec_{C-Reduce-slow} spends 1.40% and 1.36% more time than C-Reduce-slow on reducing crash test cases and miscompilation test cases, respectively.

Comparing with DDSET. As shown in Table 4.1, DDSET cannot eliminate more crash test cases than Perses. Moreover, its average execution time is $8.04\times$ that of Perses. For DDSET, its results contain abstract symbols (*i.e.*, terminals and nonterminals), which makes it impossible to count tokens. Considering that token count is not the focus of this experiment, and it is not clear what is the proper way to count abstract symbols, we did not include the token count results for DDSET. In terms of reducing miscompilation

test cases, as shown in Table 4.2, DDSET cannot eliminate any test case, and its average execution time is the highest among all the reducers. It should be noted that DDSET exceeds the time limit (set to 7,200 seconds) for 897 test cases, and the results shown in the table do not include these timeout results. To understand the reason why DDSET is not effective in this task, we investigated a few results produced by DDSET and found that the structures that DDSET can abstract are very limited. A possible explanation is that these C programs are too complex for DDSET to abstract. In the DDSET paper, the benchmark suite used for evaluation includes four input languages, *i.e.*, JavaScript, Clojure, Lua, and UNIX command line utilities including grep and find, and the bug-triggering inputs after the reduction by Perses (with which the abstracting process starts) are relatively small (46.82 characters on average and 185 characters at most). In contrast, the crash test cases reduced by Perses contains 244.94 bytes on average and 3,080 bytes at most. Moreover, the restrictions imposed upon the elements of a C test case by the strict semantic requirements and the property (*i.e.*, triggering the bug) may make the abstracting process infeasible.

RQ1: T-Rec can enable previous reducers to eliminate more duplicates by reducing them to textually equivalent test cases. Specifically, T-Rec_{Perses}, T-Rec_{Vulcan}, T-Rec_{C-Reduce}, and T-Rec_{C-Reduce-slow} eliminate 1,294, 1,315, 336, and 128 more duplicates (including eliminated crash test cases and miscompilation test cases) than Perses, Vulcan, C-Reduce, and C-Reduce-slow, respectively.

4.5.2 RQ2: Effectiveness in Reducing Program Size in Bytes

The first research question has already demonstrated that T-Rec is effective in deduplication, which indicates that T-Rec is effective in canonicalization to some extent. In this research question, we evaluate the canonicalization capability of T-Rec from another angle. Since the canonical token in this work is defined with shortlex order, the byte size of the reduced program reflects the degree of canonicalization to some extent. A reducer with a higher degree of canonicalization is expected to produce results with smaller size in bytes. Therefore, in this experiment, we measure the program sizes in bytes of the results produced by each tool on **Benchmark-Reduce**. The results are shown in Table 4.3 and Figure 4.8.

T-Rec_{Perses} vs. C-Reduce-canop_{Perses} vs. Perses. In this experiment, T-Rec_{Perses} significantly outperforms both Perses and C-Reduce-canop_{Perses}. In 199 out of 245 benchmarks, the result of T-Rec_{Perses} is smaller than both that of Perses and C-Reduce-canop_{Perses}, and in the remaining benchmarks, the results of T-Rec_{Perses} and C-Reduce-canop_{Perses} have

Table 4.3: The program sizes in bytes of the results produced by different reducers on C and Rust benchmarks. CR represents C-Reduce-cano.

Subjects	Perses	CR _{Perses}	T-Rec _{Perses}	Change <i>w.r.t.</i> Perses	Change <i>w.r.t.</i> CR _{Perses}	Vulcan	CR _{Vulcan}	T-Rec _{Vulcan}	Change <i>w.r.t.</i> Vulcan	Change <i>w.r.t.</i> CR _{Vulcan}
clang-22382	524	285	231	-55.92%	-18.95%	344	189	189	-45.06%	0.00%
clang-22704	246	124	117	-52.44%	-5.65%	193	94	94	-51.30%	0.00%
clang-23309	1,899	716	153	-91.94%	-78.63%	1,237	465	431	-65.16%	-7.31%
clang-23353	331	169	152	-54.08%	-10.06%	308	150	150	-51.30%	0.00%
clang-25900	926	412	328	-64.58%	-20.39%	353	149	149	-57.79%	0.00%
clang-26760	487	193	115	-76.39%	-40.41%	183	103	104	-43.17%	0.97%
clang-27137	684	308	80	-88.30%	-74.03%	354	141	126	-64.41%	-10.64%
clang-27747	415	193	128	-69.16%	-33.68%	301	116	116	-61.46%	0.00%
clang-31259	1,341	617	424	-68.38%	-31.28%	843	390	359	-57.41%	-7.95%
gcc-59903	906	464	446	-50.77%	-3.88%	518	277	276	-46.72%	-0.36%
gcc-60116	1,901	764	455	-76.07%	-40.45%	739	387	344	-53.45%	-11.11%
gcc-61383	980	416	308	-68.57%	-25.96%	649	264	265	-59.17%	0.38%
gcc-61917	557	250	199	-64.27%	-20.40%	319	144	145	-54.55%	0.69%
gcc-64990	733	436	314	-57.16%	-27.98%	553	271	270	-51.18%	-0.37%
gcc-65383	516	223	186	-63.95%	-16.59%	241	115	119	-50.62%	3.48%
gcc-66186	1,128	473	396	-64.89%	-16.28%	668	288	290	-56.59%	0.69%
gcc-66375	1,596	671	491	-69.24%	-26.83%	824	335	318	-61.41%	-5.07%
gcc-70127	1,000	503	339	-66.10%	-32.60%	768	334	317	-58.72%	-5.09%
gcc-70586	633	238	221	-65.09%	-7.14%	326	157	158	-51.53%	0.64%
gcc-71626	167	105	95	-43.11%	-9.52%	137	91	91	-33.58%	0.00%
Average	848.5	378	258.9	-65.52%	-27.04%	492.9	223	215.55	-53.73%	-2.05%
rust-111502	429	299	292	-31.93%	-2.34%	395	284	282	-28.61%	-0.70%
rust-112061	1,002	951	727	-27.45%	-23.55%	963	912	757	-21.39%	-17.00%
rust-112213	1,509	1,335	1,068	-29.22%	-20.00%	1,323	1,171	1,062	-19.73%	-9.31%
rust-112526	996	984	804	-19.28%	-18.29%	900	888	726	-19.33%	-18.24%
rust-44800	1,240	850	759	-38.79%	-10.71%	795	569	524	-34.09%	-7.91%
rust-66851	1,767	1,191	1,126	-36.28%	-5.46%	1,791	1,294	1,170	-34.67%	-9.58%
rust-69039	377	274	274	-27.32%	0.00%	295	267	259	-12.20%	-3.00%
rust-77002	452	368	328	-27.43%	-10.87%	468	380	360	-23.08%	-5.26%
rust-77320	84	73	73	-13.10%	0.00%	86	78	74	-13.95%	-5.13%
rust-77323	49	49	49	0.00%	0.00%	49	49	49	0.00%	0.00%
rust-77910	62	53	43	-30.65%	-18.87%	43	39	37	-13.95%	-5.13%
rust-77919	176	135	135	-23.30%	0.00%	172	141	132	-23.26%	-6.38%
rust-78005	188	157	157	-16.49%	0.00%	218	209	157	-27.98%	-24.88%
rust-78325	95	63	63	-33.68%	0.00%	92	60	60	-34.78%	0.00%
rust-78651	53	34	34	-35.85%	0.00%	24	24	24	0.00%	0.00%
rust-78652	230	139	134	-41.74%	-3.60%	141	127	122	-13.48%	-3.94%
rust-78655	50	41	41	-18.00%	0.00%	49	43	41	-16.33%	-4.65%
rust-78720	174	100	100	-42.53%	0.00%	107	86	86	-19.63%	0.00%
rust-91725	338	255	156	-53.85%	-38.82%	208	161	155	-25.48%	-3.73%
rust-99830	685	623	548	-20.00%	-12.04%	632	580	544	-13.92%	-6.21%
Average	497.8	398.7	345.55	-28.34%	-8.23%	437.55	368.1	331.05	-19.79%	-6.55%

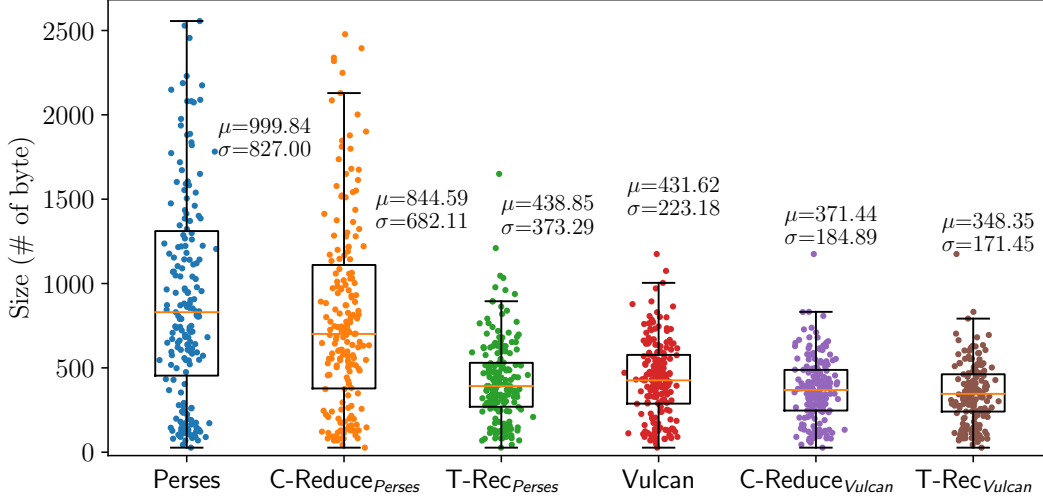


Figure 4.8: The program sizes in bytes of the results produced by different reducers on SMT-LIBv2 benchmarks. On average, the results of T-Rec_{Perses} are 42.86% and 35.39% smaller than Perses and C-Reduce-cano_{Perses}, respectively; the results of T-Rec_{Vulcan} are 16.24% and 4.83% smaller than Vulcan and C-Reduce-cano_{Vulcan}, respectively.

the same sizes. On average, the result of T-Rec_{Perses} is 65.52% (and 27.04%), 28.34% (and 8.23%), and 42.86% (and 35.39%) smaller than that of Perses (and C-Reduce-cano_{Perses}) on C, Rust, and SMT-LIBv2 benchmarks, respectively. It should be noted that part of the decrease in byte size achieved by T-Rec_{Perses} is attributed to the fact that T-Rec_{Perses} removes more tokens than Perses.

T-Rec_{Vulcan} vs. Vulcan. Compared to Perses, the reduction algorithm of Vulcan is more exhaustive, and T-Rec_{Vulcan} fails to remove more tokens than Vulcan (as shown in § 4.5.3). Nevertheless, T-Rec_{Vulcan} still manages to produce 53.73%, 19.79%, and 16.24% smaller outcomes than Vulcan on C, Rust, and SMT-LIBv2 benchmarks, respectively. Such a result indicates that T-Rec is effective in increasing the degree of canonicalization.

T-Rec_{Vulcan} vs. C-Reduce-cano_{Vulcan}. Although C-Reduce-cano_{Vulcan} is also effective in terms of increasing the degree of canonicalization, T-Rec_{Vulcan} significantly outperforms C-Reduce-cano_{Vulcan} on Rust and SMT-LIBv2 benchmarks. The p-values yielded by the Wilcoxon signed-rank test [217] are 4.34×10^{-4} and 1.36×10^{-4} , respectively. This result demonstrates that T-Rec has better generality than C-Reduce-cano. On C benchmarks, T-Rec_{Vulcan} performs slightly better than C-Reduce-cano_{Vulcan}, but the improvement is not

significant. The p-value is 0.14. We also noticed that in 6 out of 20 C benchmarks and in 7 out of 205 SMT-LIBv2 benchmarks, the results of C-Reduce-cano_{Vulcan} are smaller than those of T-Rec_{Vulcan}, and we manually investigated each of them. For the 6 C benchmarks, the differences are due to C-Reduce sometimes reducing an integer to a smaller size in bytes than T-Rec (*e.g.*, it can convert a hex to a decimal, thus removing `0x`). For the 7 SMT-LIBv2 benchmarks, the differences are all due to T-Rec considering simple symbols as keywords (according to the lexical syntax) and thus not reducing them, whereas the passes of C-Reduce treat them as identifiers and replace them with smaller identifiers (taking fewer bytes).

RQ2: Both T-Rec_{Perses} and T-Rec_{Vulcan} exhibit significantly better capability in canonicalization than Perses and Vulcan, respectively. Specifically, on average, the results of T-Rec_{Perses} (and T-Rec_{Vulcan}) are 65.52% (53.73%), 28.34% (19.79%), and 42.86% (16.24%) smaller than that of Perses (and Vulcan) on C, Rust, and SMT-LIBv2 benchmarks, respectively. Although integrating the passes for canonicalization in C-Reduce to Perses and Vulcan also increases the degree of canonicalization, such an approach is not as generalized as T-Rec and is significantly outperformed by T-Rec on Rust and SMT-LIBv2 benchmarks.

4.5.3 RQ3: Effectiveness in Removing Tokens

The evaluation results with **Benchmark-Reduce** also demonstrate that T-Rec_{Perses} significantly outperforms Perses and C-Reduce-cano_{Perses} in terms of removing tokens. Table 4.4 and Figure 4.9 show the token counts of the results produced by each reducer.

On average, the results of T-Rec_{Perses} contain 21.30%, 4.71%, and 34.35% fewer tokens than that of Perses. In 185 (20 C, 8 Rust, and 157 SMT-LIBv2) out of 245 benchmarks, T-Rec_{Perses} reduces more tokens than Perses. Although C-Reduce-cano_{Perses} also reduces more tokens than Perses in some benchmarks, it only achieves this in 15 (1 C and 14 SMT-LIBv2) out of 245 benchmarks, and there is no benchmark where C-Reduce-cano_{Perses} reduces more tokens than T-Rec_{Perses}.

T-Rec_{Vulcan} does not remove noticeably more tokens than Vulcan overall. Specifically, the results of T-Rec_{Vulcan} on C and Rust benchmarks contain the same number of tokens as those of Vulcan. Such a result is expected to some extent since the reduction algorithm of Vulcan is much more aggressive than Perses. Compared to Perses, it contains three additional auxiliary reducers to perform transformations for exploring reduction opportunities. Nevertheless, T-Rec_{Vulcan} still manages to remove more tokens in 18 SMT-LIBv2

Table 4.4: The token counts of the results produced by different reducers on C and Rust. CR represents C-Reduce-cano.

Subjects	Perses	CR _{Perses}	T-Rec _{Perses}	Change <i>w.r.t.</i> Perses	Vulcan	CR _{Vulcan}	T-Rec _{Vulcan}	Change <i>w.r.t.</i> Vulcan
clang-22382	144	144	129	-10.42%	108	108	108	0.00%
clang-22704	78	78	71	-8.97%	62	62	62	0.00%
clang-23309	464	464	102	-78.02%	303	303	303	0.00%
clang-23353	98	98	94	-4.08%	91	91	91	0.00%
clang-25900	239	239	211	-11.72%	104	104	104	0.00%
clang-26760	120	120	74	-38.33%	56	56	56	0.00%
clang-27137	180	180	47	-73.89%	88	88	88	0.00%
clang-27747	117	117	89	-23.93%	79	79	79	0.00%
clang-31259	406	406	331	-18.47%	282	282	282	0.00%
gcc-59903	308	308	304	-1.30%	198	198	198	0.00%
gcc-60116	488	488	331	-32.17%	241	241	241	0.00%
gcc-61383	272	272	221	-18.75%	195	195	195	0.00%
gcc-61917	150	150	134	-10.67%	103	103	103	0.00%
gcc-64990	239	239	221	-7.53%	203	203	203	0.00%
gcc-65383	153	153	134	-12.42%	84	84	84	0.00%
gcc-66186	327	327	296	-9.48%	226	226	226	0.00%
gcc-66375	440	440	368	-16.36%	227	227	227	0.00%
gcc-70127	301	301	240	-20.27%	230	230	230	0.00%
gcc-70586	157	153	142	-9.55%	94	94	94	0.00%
gcc-71626	51	51	41	-19.61%	38	38	38	0.00%
Average	236.6	236.4	179	-21.30%	150.6	150.6	150.6	0.00%
rust-111502	166	166	161	-3.01%	157	157	157	0.00%
rust-112061	458	458	413	-9.83%	442	442	442	0.00%
rust-112213	736	736	647	-12.09%	635	635	635	0.00%
rust-112526	382	382	382	0.00%	338	338	338	0.00%
rust-44800	467	467	467	0.00%	284	284	284	0.00%
rust-66851	728	728	724	-0.55%	713	713	713	0.00%
rust-69039	114	114	114	0.00%	101	101	101	0.00%
rust-77002	263	263	247	-6.08%	247	247	247	0.00%
rust-77320	39	39	39	0.00%	39	39	39	0.00%
rust-77323	13	13	13	0.00%	13	13	13	0.00%
rust-77910	34	34	28	-17.65%	21	21	21	0.00%
rust-77919	74	74	74	0.00%	62	62	62	0.00%
rust-78005	102	102	102	0.00%	102	102	102	0.00%
rust-78325	28	28	28	0.00%	26	26	26	0.00%
rust-78651	17	17	17	0.00%	9	9	9	0.00%
rust-78652	56	56	56	0.00%	49	49	49	0.00%
rust-78655	26	26	26	0.00%	26	26	26	0.00%
rust-78720	72	72	72	0.00%	56	56	56	0.00%
rust-91725	174	174	105	-39.66%	105	105	105	0.00%
rust-99830	299	299	283	-5.35%	277	277	277	0.00%
Average	212.4	212.4	199.9	-4.71%	185.1	185.1	185.1	0.00%

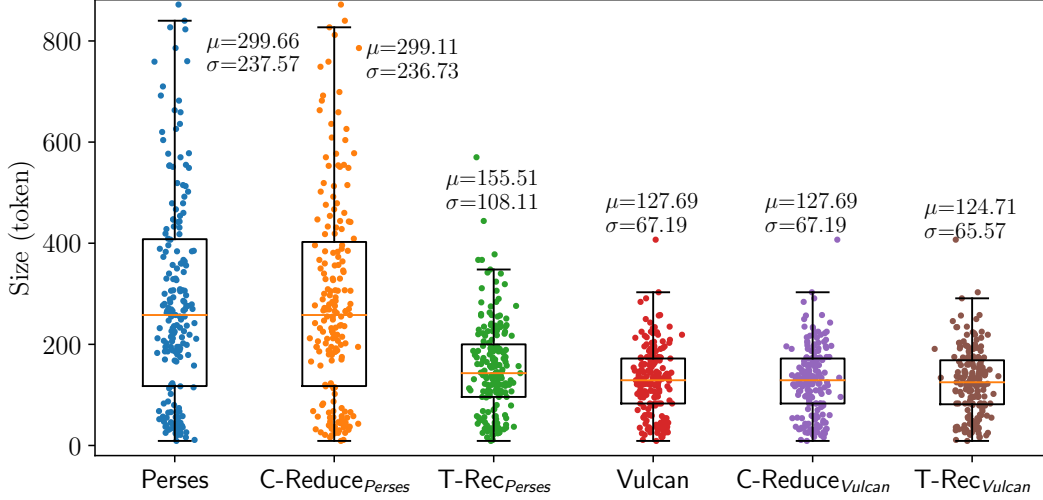


Figure 4.9: The token counts of the results produced by different reducers on SMT-LIBv2 benchmarks. On average, the results of T-Rec_{Perses} contain 34.35% fewer tokens than Perses.

benchmarks. Moreover, in these 18 benchmarks, the results of T-Rec_{Vulcan} contain 17.70% fewer tokens on average, and in the extreme case, the result of T-Rec_{Vulcan} contains 63.13% fewer tokens. On the other hand, C-Reduce-cano_{Vulcan} cannot remove more tokens than Vulcan in any benchmark.

RQ3: T-Rec_{Perses} significantly outperforms Perses and C-Reduce-cano_{Perses} in terms of removing tokens. On average, the result of T-Rec_{Perses} contains 21.30%, 4.71%, and 34.35% fewer tokens than that of Perses on C, Rust, and SMT-LIBv2 benchmarks, respectively. Additionally, T-Rec_{Vulcan} can remove more tokens than Vulcan in 18 out of 205 SMT-LIBv2 benchmarks despite the aggressive reduction algorithm adopted by Vulcan.

4.5.4 RQ4: Efficiency

Since T-Rec_{Perses} and T-Rec_{Vulcan} perform extra steps on top of Perses and Vulcan, it is inevitable that T-Rec_{Perses} and T-Rec_{Vulcan} take more time to reduce a program. The experiment discussed in § 4.5.1 has demonstrated this to some extent. This research

Table 4.5: The execution time of different reducers on C and Rust. CR represents C-Reduce-cano.

Subjects	Perses	CR _{Perses}	T-Rec _{Perses}	Change w.r.t. Perses	Change w.r.t. CR _{Perses}	Vulcan	CR _{Vulcan}	T-Rec _{Vulcan}	Change w.r.t. Vulcan	Change w.r.t. CR _{Vulcan}
clang-22382	256	267	266	3.91%	-0.37%	525	539	534	1.71%	-0.93%
clang-22704	609	617	630	3.45%	2.11%	726	738	736	1.38%	-0.27%
clang-23309	882	964	934	5.90%	-3.11%	5,012	5,113	5,036	0.48%	-1.51%
clang-23353	320	330	329	2.81%	-0.30%	451	462	452	0.22%	-2.16%
clang-25900	388	411	429	10.57%	4.38%	848	866	859	1.30%	-0.81%
clang-26760	806	823	842	4.47%	2.31%	1,136	1,150	1,153	1.50%	0.26%
clang-27137	4,023	4,152	4,122	2.46%	-0.72%	7,027	7,148	7,085	0.83%	-0.88%
clang-27747	509	531	557	9.43%	4.90%	869	892	879	1.15%	-1.46%
clang-31259	1,013	1,343	1,927	90.23%	43.48%	9,553	9,780	9,639	0.90%	-1.44%
gcc-59903	1,822	1,872	1,898	4.17%	1.39%	2,677	2,718	2,695	0.67%	-0.85%
gcc-60116	1,252	1,442	1,718	37.22%	19.14%	6,126	6,208	6,161	0.57%	-0.76%
gcc-61383	1,670	1,876	2,141	28.20%	14.13%	9,766	9,986	9,859	0.95%	-1.27%
gcc-61917	559	572	602	7.69%	5.24%	807	819	815	0.99%	-0.49%
gcc-64990	1,502	1,553	1,798	19.71%	15.78%	3,003	3,077	3,126	4.10%	1.59%
gcc-65383	522	571	594	13.79%	4.03%	1,013	1,073	1,036	2.27%	-3.45%
gcc-66186	1,674	1,997	2,521	50.60%	26.24%	15,942	16,305	16,560	3.88%	1.56%
gcc-66375	1,763	2,097	3,344	89.68%	59.47%	10,432	10,699	10,558	1.21%	-1.32%
gcc-70127	1,949	2,310	2,969	52.33%	28.53%	13,632	13,938	13,818	1.36%	-0.86%
gcc-70586	3,317	3,830	4,214	27.04%	10.03%	15,143	15,893	15,348	1.35%	-3.43%
gcc-71626	24	26	29	20.83%	11.54%	40	43	42	5.00%	-2.33%
Average	1,243	1,379.2	1,593.2	24.22%	12.41%	5,236.4	5,372.35	5,319.55	1.59%	-1.04%
rust-111502	39	63	49	25.64%	-22.22%	275	292	283	2.91%	-3.08%
rust-112061	2,010	3,305	3,418	70.05%	3.42%	12,565	13,834	13,605	8.28%	-1.66%
rust-112213	3,235	5,287	3,649	12.80%	-30.98%	36,784	38,757	37,682	2.44%	-2.77%
rust-112526	2,643	3,112	5,064	91.60%	62.72%	7,799	8,237	9,673	24.03%	17.43%
rust-44800	456	830	575	26.10%	-30.72%	2,268	2,415	2,378	4.85%	-1.53%
rust-66851	3,170	5,500	4,771	50.50%	-13.25%	20,395	22,603	22,233	9.01%	-1.64%
rust-69039	437	584	571	30.66%	-2.23%	3,875	4,147	3,875	0.00%	-6.56%
rust-77002	191	235	189	-1.05%	-19.57%	616	651	653	6.01%	0.31%
rust-77320	7	11	7	0.00%	-36.36%	52	58	54	3.85%	-6.90%
rust-77323	4	6	4	0.00%	-33.33%	9	11	9	0.00%	-18.18%
rust-77910	7	11	9	28.57%	-18.18%	27	29	30	11.11%	3.45%
rust-77919	13	24	16	23.08%	-33.33%	96	103	100	4.17%	-2.91%
rust-78005	9	9	13	44.44%	44.44%	72	73	74	2.78%	1.37%
rust-78325	2	5	3	50.00%	-40.00%	20	21	20	0.00%	-4.76%
rust-78651	5	7	4	-20.00%	-42.86%	17	20	16	-5.88%	-20.00%
rust-78652	7	13	11	57.14%	-15.38%	58	62	57	-1.72%	-8.06%
rust-78655	2	4	3	50.00%	-25.00%	19	36	21	10.53%	-41.67%
rust-78720	12	23	16	33.33%	-30.43%	116	123	116	0.00%	-5.69%
rust-91725	649	676	783	20.65%	15.83%	1,098	1,112	1,118	1.82%	0.54%
rust-99830	6,779	7,566	8,878	30.96%	17.34%	36,539	37,060	42,597	16.58%	14.94%
Average	983.85	1,363.55	1,401.65	31.22%	-12.51%	6,135	6,482.2	6,729.7	5.04%	-4.37%

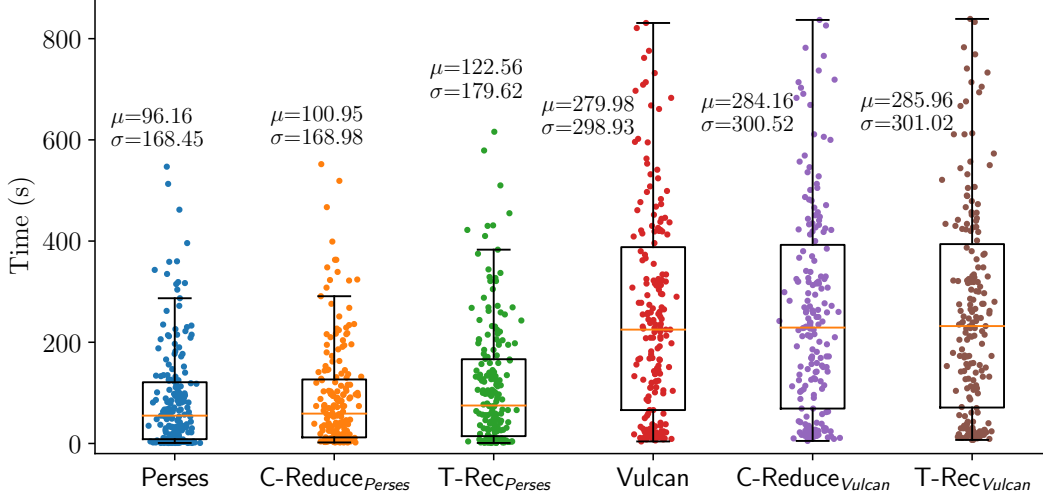


Figure 4.10: The execution time of different reducers on SMT-LIBv2 benchmarks. On average, T-Rec_{Perses} takes 56.46% and 22.04% more time than Perses and C-Reduce-cano_{Perses}, respectively; T-Rec_{Vulcan} takes 5.01% more time than Vulcan.

question aims to further investigate how much overhead is brought by the additional fine-grained reduction process in each variant of T-Rec using **Benchmark-Reduce**. To answer this research question, we compare the execution time that each tool takes to reduce the benchmark programs. The results are shown in Table 4.5 and Figure 4.10.

Perses and Vulcan as the Baselines. Overall, T-Rec_{Perses} takes 24.22%, 31.22%, and 56.46% more time than Perses on C, Rust, and SMT-LIBv2 benchmarks, respectively. Compared to Vulcan, which takes 265.37%, 539.89%, and 537.53% more time than Perses on C, Rust, and SMT-LIBv2 benchmarks, the overhead introduced by the fine-grained reduction process in T-Rec_{Perses} is moderate, and does not make T-Rec impractical. Additionally, the overhead of applying T-Rec to Vulcan is more marginal than that of applying it to Perses. Specifically, T-Rec_{Vulcan} takes 1.59%, 5.04%, and 5.01% more time than Vulcan on C, Rust, and SMT-LIBv2 benchmarks, respectively.

C-Reduce-cano_{Perses} and C-Reduce-cano_{Vulcan} as the Baselines. Compared to C-Reduce-cano_{Perses}, T-Rec_{Perses} takes significantly more time than C-Reduce-cano_{Perses} on C and SMT-LIBv2 benchmarks (*i.e.*, 12.41% and 22.04% more time on average) and does not show significant efficiency difference on Rust benchmarks (p-value is 0.28). The reason behind this is that T-Rec is more effective in removing more tokens than C-Reduce-cano is.

Therefore, to reach a fixed point, more rounds of alternately invoking the syntax-guided reduction algorithm (*i.e.*, `Perses`) and the fine-grained reduction algorithm are required. The fact that `T-RecVulcan` takes a similar execution time as `C-Reduce-canoVulcan` further verifies such an explanation.

RQ4: The overhead brought by the fine-grained reduction process in T-Rec is reasonable. Specifically, `T-RecPerses` takes 24.22%, 31.22%, and 56.46% more time than `Perses` on C, Rust, and SMT-LIBv2 benchmarks, respectively, and `T-RecVulcan` takes 1.59%, 5.04%, and 5.01% more time than `Vulcan` on each benchmark.

4.6 Discussion

In this section, we discuss an ablation study we conducted to further evaluate the fine-grained reduction process of T-Rec and the potential threats to the validity of this work.

4.6.1 Comparing to Perses with Lex Tree Extension

A straightforward way to perform fine-grained program reduction is to first extend the parse tree of the input program with lex trees (*i.e.*, convert leaf nodes to corresponding lex trees), and then directly utilize a tree-based syntax guided program reduction algorithm, *e.g.*, `Perses`, to reduce the extended parse tree. Compared to such an approach, the fine-grained reduction process of T-Rec contains extra canonicalization steps. As introduced in § 4.4, T-Rec performs replacement-based token canonicalization and deletion-based token canonicalization, and deletion-based token canonicalization can be further divided into two sub-steps, *i.e.*, lexical syntax-guided reduction and lex tree-based canonicalization. Among all these steps, lexical syntax-guided reduction can be considered as directly applying the `Perses` algorithm to lex tree.³ To evaluate the effectiveness of the other two extra steps that T-Rec performs, we conducted an ablation study that compares the deduplication performance of integrating T-Rec and T-Rec-reduce-only (a variant of T-Rec that only performs lexical syntax-guided reduction in the fine-grained reduction process) to `Perses`, `Vulcan`, `C-Reduce`, and `C-Reduce-slow` using the 2,501 crash test cases in **Benchmark-Cano**. The results presented in Table 4.6 show that compared to reducers integrated with T-Rec

³There is a slight difference between lexical syntax-guided reduction and directly applying the `Perses` to the parse tree extended with lex trees. Recall that when T-Rec attempts to reduce a token, it also attempts to reduce all the other identical tokens in the same way.

(*i.e.*, T-Rec_{Perses}, T-Rec_{Vulcan}, T-Rec_{C-Reduce}, T-Rec_{C-Reduce-slow}), the corresponding reducers with T-Rec-reduce-only integrated eliminate 248, 240, 326, 223 fewer crash C test cases, indicating the extra steps performed by T-Rec significantly contribute to the high capability of canonicalization.

Table 4.6: The reduction results produced by different reducers on the 2,501 crash test cases in **Benchmark-Cano**. The column **Eliminated (#)** shows the total number of programs eliminated by each reducer.

Reducer	Eliminated (#)		
	With T-Rec integrated	With T-Rec-reduce-only integrated	Difference
Perses	1,297	1,049	248
Vulcan	1,327	1,087	240
C-Reduce	1,466	1,140	326
C-Reduce-slow	2,053	1,830	223

4.6.2 Threats to Validity

The correctness of the implementation of T-Rec is vital to the internal validity. To mitigate the threats stemming from this aspect, we checked whether the programs reduced by T-Rec could still pass the property test. In addition, to prevent bug slippage (*i.e.*, cases where the program triggers a different bug after reduction), the property tests used in our benchmarks are designed to be comprehensive. Furthermore, when counting the number of eliminated duplicates, we compute the count separately for each group of programs that are known to originally trigger the same bug, and then sum these counts, rather than counting programs that become identical across different bug groups. The nondeterminism in some of the benchmarks is also a threat to the internal validity. Sometimes, it is hard to promise that the bug is triggered in each compilation process. To mitigate this threat, first we manually patched the benchmarks that were nondeterministic. Specifically, we revised the corresponding property test script to repeat the test multiple times to decrease the degree of nondeterminism. Second, all the experiments except the deduplication experiment (§ 4.5.1) are run three times, and we report the average over the multiple runs. For the deduplication experiment, we did not run it multiple times for all the reducers because reducers like C-Reduce-slow and T-Rec_{C-Reduce-slow} take a considerable amount of time to finish. However, we did run some efficient reducers multiple times and we observed negligible differences

among the different runs. Another possible threat to validity comes from the measurement of program size in bytes. The format of the program can affect the program size of the program significantly. To mitigate this threat, we do not count any whitespace characters when measuring the sizes.

To mitigate the threats to external validity, *i.e.*, the generality of T-Rec’s evaluation results, we collected benchmarks from previous research studies [192, 114, 221, 200], which cover three different programming languages. Moreover, these benchmarks are diverse in terms of the size of the original bug-triggering program (ranging from 81 bytes to 793470 bytes) and bug type (crash bugs and miscompilation bugs).

4.7 Related Work

In this section, we introduce and discuss some research studies that are closely related to T-Rec.

4.7.1 Language-Agnostic Program Reducers

The first work of program reduction is Delta Debugging (DD), proposed in 2002. As a groundbreaking work, DD inspired many follow-up research studies in the field of failure-inducing input reduction [226]. Given a failure-inducing input, the proposed minimizing delta debugging algorithm `ddmin` systematically removes the failure-irrelevant elements in the input until removing any single element from the input cannot trigger the same failure. To improve the performance of DD, a recent study proposed Probabilistic Delta Debugging (ProbDD), which uses a probabilistic model to guide the removal of elements [209].

However, as DD treats inputs as flat lists, it can hardly handle excessively large and highly structured inputs like programs. To overcome this problem, Hierarchical Delta Debugging (HDD) converts inputs to corresponding tree structures (*e.g.*, parse trees) and applies `ddmin` on each level of the tree in a top-down manner [152]. In this way, reduction starts from the coarsest granularity, gradually becomes fine-grained, and can also recognize the structure of the input; thus both efficiency and effectiveness are enhanced. A few subsequent studies improve HDD in various ways. Picireny improves the performance of HDD by using extended context-free grammar [95]. Coarse Hierarchical Delta Debugging (CHDD) significantly reduces the number of property tests performed during the reduction process without noticeably deteriorating the reduction quality [97]. HDDr, a recursive

variant of HDD [112], significantly reduces the time required for reduction. There is another work that improves HDD by extending it with a technique called hoisting [208].

Since HDD leverages the formal grammar only to build the tree structure, it does not guarantee that the generated programs are syntactically valid. As a result, many syntactically invalid inputs are generated and tested against the property during the reduction process. Since such invalid inputs usually do not trigger the same failure as the original input, the efficiency of HDD is restricted. To address this, Sun *et al.* proposed Perses, a syntax-guided reduction approach [192]. By changing the way to use `ddmin` and designing a hoisting transformation, it avoids generating syntactically invalid inputs when trying to delete components from the program. Vulcan is a recently proposed reduction framework [221]. It contains a main reducer and utilizes a list of auxiliary reducers to help produce smaller results. Specifically, it first minimizes the program with the main reducer. Whenever the main reducer reaches a local minimum, one of the auxiliary reducers is invoked to either perform transformations or a more exhaustive search to possibly escape the local minimum, after which the reduction can be continued by the main reducer. Tian *et al.* further proposed *Refreshable Compact Cache*, a domain-specific caching scheme to speed up the program reduction [200].

However, all these mentioned tree-based approaches are limited by their coarse reduction granularity. All of them treat a token as an atomic element, and thus can only entirely remove only entire tokens or keep the program as is. In contrast, T-Rec can reduce tokens with a two-stage lexical syntax-guided reduction algorithm. Such a finer-grained approach enables T-Rec to achieve smaller reduction results while not introducing an impractical overhead.

DDSET pursues a high degree of canonicalization with a different approach [83]. It strives to abstract syntactical structures of the reduced input. In other words, it attempts to convert some concrete structures in the input into generalized abstract symbols (*i.e.*, terminals and nonterminals). However, although DDSET is demonstrated to be effective with a benchmark suite including JavaScript, Clojure, Lua and UNIX command line utilities inputs, according to our evaluation, it is not as effective and efficient as T-Rec in deduplicating bug-triggering C programs. One possible explanation is that when the input has strict restrictions imposed upon its tokens and syntactical structures by the semantics or the property (*i.e.*, triggering a certain bug), the abstraction process may become infeasible, causing the effectiveness of DDSET to deteriorate.

Another test case reduction technique that is related to this work is internal test case reduction [144]. This approach also adopts shortlex order to pursuit both reduction and canonicalization, but it performs reduction in a very different manner. Instead of directly

performing reduction on the test case, it reduces the choice sequence that leads a test case generator to produce the test case under reduction. The advantage of internal test case reduction is that it can equip each test case generator with reduction ability for free, without the need of an external reducer. Meanwhile, it can benefit from certain properties that the generator has. For example, if a generator is crafted to generate only valid test cases, then internal reduction can promise the intermediate test cases generated during the reduction process are all valid, thus avoiding wasting time on checking invalid test cases. Compared to internal test case reduction, T-Rec does not rely on the quality or presence of a test case generator, and we believe that external reduction techniques like T-Rec and internal reduction are complementary to each other.

4.7.2 Language-Specific Program Reducers

Besides the language-agnostic reduction approaches mentioned above, there are also many language-specific reduction tools. These tools are specifically optimized for certain languages. Therefore, they are more effective in reducing programs that are written in those specific languages. The downside of such tools is that these tools typically require domain-specific knowledge and heavy engineering efforts to build. C-Reduce is a program reduction tool specifically optimized for reducing C/C++ programs. It is implemented with a set of C/C++-specific transformations based on the Clang front end to help reduce the programs [170]. ddSMT and ddSMT2.0 are designed for reducing SMT-LIBv2 programs [161, 114]. ddSMT is based on **ddmin**, and it adds a set of SMT-LIBv2-specific structural and semantic simplifications. ddSMT2.0 further improves ddSMT by supporting the entire family of SMT-LIBv2 language dialects and providing a hierarchical reduction strategy in addition to the original **ddmin**-based strategy. JS Delta is a JavaScript-specific reducer that reduces JavaScript programs by deleting statements, functions, and sub-expressions [108]. J-Reduce is a reducer for reducing Java bytecode [109]. It is implemented with a general strategy to overcome the challenge in reducing dependency graphs. A following work improves J-Reduce by using a more fine-grained dependency modeling approach [110]. CHISEL is designed for debloating C/C++ programs, which utilizes reinforcement learning to improve the performance [92].

Some of the aforementioned language-specific program reducers recognize the importance of reducing tokens to a canonical form, and are implemented with such functionality. For example, C-Reduce can refactor a program by renaming all the identifiers with shorter and canonical names (*i.e.*, **a**, **b**, **c**, $\dots$, **aa**, **ab**, **ac**, $\dots$), removing the content of each string literal in the program, and replacing integer tokens with shorter ones by using regular expression substitutions. However, such techniques are not generalized enough, and also they

are not likely to enable further deletion of tokens. In contrast, T-Rec can canonicalize tokens for any programming language as it leverages the lexical syntax, and is demonstrated to be effective in terms of creating reduction opportunities for removing more tokens.

4.8 Chapter Conclusion

In this chapter, we pointed out that the performance of previous language-agnostic program reduction approaches is restricted by their coarse reduction granularity. Because they treat tokens as the fundamental elements of a program, previous approaches can remove only an entire token or keep it as is during reduction, but cannot shrink or canonicalize tokens. We propose T-Rec, a fine-grained language-agnostic program reduction technique. By guiding the fine-grained reduction process with the lexical syntax, T-Rec can effectively and efficiently enhance the reduction and canonicalization capability of existing language-agnostic program reducers. Our extensive evaluation with versatile benchmark suites demonstrates the effectiveness and efficiency of T-Rec. Specifically, T-Rec enables Perses, Vulcan, C-Reduce, and C-Reduce-slow to further eliminate 1,294, 1,315, 336, and 128 duplicates in a benchmark suite that contains 3,796 test cases triggering 46 bugs in GCC. The evaluation also demonstrates that T-Rec is more effective and efficient than DDSET. Additionally, T-Rec further reduces 65.52% (53.73%), 28.34% (19.79%), and 42.86% (16.24%) bytes of the results of Perses (Vulcan) at the cost of 24.22% (1.59%), 31.22% (5.04%), and 56.46% (5.01%) extra execution time on C, Rust, and SMT-LIBv2 benchmarks, respectively. Moreover, T-Rec also further reduces 21.30%, 4.71%, and 34.35% tokens in the results of Perses on each language. By comparing to the approach used in C-Reduce, we also demonstrate that T-Rec has better generality and can create more reduction opportunities for removing more tokens.

Chapter 5

Boosting Program Reduction with the Missing Piece of Syntax-Guided Transformations

This chapter introduces SFC, a new syntax-guided transformation, along with three reduction methods designed to utilize it effectively. These SFC-based techniques enrich the search space, enabling additional structural simplifications and canonicalization. When incorporated into Vulcan alongside T-Rec, these methods contribute further size reductions and eliminate additional duplicates, underscoring their effectiveness in enhancing both minimization and canonicalization.

5.1 Introduction

Program reduction presents significant challenges for two primary reasons. First, the original bug-triggering programs can be considerably large since many of these programs are generated through fuzz testing [189, 223, 117], and empirical evidence suggests that configuring a fuzzer to generate larger programs can enhance bug-finding performance [223]. Meanwhile, a language processor is typically extremely complex, and thus reducing the program by directly analyzing which part is essential for triggering the bug is almost impossible. Second, programs are highly structured and adhere to complex, strict grammatical rules. Naively removing parts of a program using algorithms like `ddmin` [226] often produces syntactically invalid code, which is unlikely to reproduce the original bug, leading to inefficient and ineffective reduction.

To address these challenges, many trial-and-error, *syntax-aware* program reduction techniques have been proposed over the past decades [152, 192, 79, 221, 170, 161, 114]. One notable example is Hierarchical Delta Debugging (HDD), proposed by Misherghe and Su [152]. HDD builds on the **ddmin** reduction algorithm but introduces a hierarchical approach: instead of directly applying **ddmin** to the program, it first transforms the program into a tree representation (*e.g.*, an abstract syntax tree) and then applies **ddmin** at each level of the tree in a top-down manner. This hierarchical strategy significantly improves both the effectiveness and efficiency of reduction in cases involving highly structured inputs. However, HDD still tends to generate a substantial number of syntactically invalid programs during the reduction process, which hampers its overall performance. This issue was addressed by Perses, a *syntax-guided* program reducer [192]. Perses additionally takes as input the formal syntax of the programming language and applies only *syntax-guided* transformations (*i.e.*, transformations that preserve the syntactical validity) during reduction. By doing so, Perses avoids generating syntactically invalid programs, leading to improved performance compared to HDD.¹

Prior Syntax-Guided Transformations. Perses introduces two syntax-guided transformations: Compatible Substructure Hoisting and Quantified Node Reduction. Compatible Substructure Hoisting replaces a given syntactical structure in a program P with one of its sub-structures (*i.e.*, a descendant of the given syntactical structure in the parse tree). For instance, it can transform the expression **a+b** into one of its sub-expressions (*e.g.*, **a** or **b**). Quantified Node Reduction uses **ddmin** to reduce syntactically parallel or repeated structures, such as a sequence of statements or a list of function parameters. These syntax-guided transformations preserve syntactical validity, have been shown to be highly effective, and have since been adopted by various program reduction techniques [221, 208, 77, 79, 230].

The Missing Piece. Despite their success, the two prior syntax-guided program transformations are inherently limited because, given a structure in program P , many smaller yet syntactically valid structures cannot be generated. Relying solely on these two transformations for program reduction excessively prunes the search space $\mathbb{P}$ and can ultimately result in inadequate reductions [221]. For instance, consider a function invocation expression **f(e1,e2,e3)** in a C program P where the arguments **e1**, **e2** and **e3** generally

¹Besides the language-agnostic reducers mentioned in the paragraph, there are many language-specific reducers such as C-Reduce and ddSMT [170, 161, 114]. These reducers are not only syntax-aware but also specifically optimized for the target languages. However, these tools require considerable language-specific knowledge and cannot be well generalized to reducing programs in other languages. In this chapter, the focus is on improving language-agnostic program reduction approaches, and thus these language-specific approaches are not discussed in detail.

represent three expressions rather than just variables, and assume that evaluating these three expressions altogether is necessary to trigger a bug, while calling `f` is irrelevant to the bug. Neither Compatible Substructure Hoisting nor Quantified Node Reduction can reduce this expression. Compatible Substructure Hoisting attempts to reduce the entire expression by replacing it with one of its sub-expressions, *i.e.*, `e1`, `e2`, or `e3`. Note that according to the C language, the syntactical structure of `e1,e2,e3` in the parse tree of calling `f` is an argument list but not an expression. Thus, Compatible Substructure Hoisting cannot replace the entire expression with `e1,e2,e3`. Quantified Node Reduction attempts to reduce the syntactically parallel structure (*i.e.*, the argument list) with the `ddmin` algorithm (*e.g.*, Quantified Node Reduction can possibly reduce the expression to `f(e1)`). Nevertheless, there exist other transformation strategies that still preserve syntactic validity. For example, deleting the function name `f` and the parentheses leaves `e1,e2,e3`, which is also a valid expression in C. Although this transformation, missed by both Compatible Substructure Hoisting and Quantified Node Reduction, only involves deleting three tokens, it can open up further reduction opportunities across the entire program P by eliminating a use of the function `f`. Namely, if `f` is unused elsewhere, its definition can also be entirely removed.

Structure Form Conversion. To mitigate the limitations of Compatible Substructure Hoisting and Quantified Node Reduction, in this chapter, we propose a novel syntax-guided transformation named Structure Form Conversion (SFC). SFC is designed to transform a given syntactic structure into other forms that preserve the syntactical validity of whole programs. For instance, given the expression `f(e1,e2,e3)`, SFC attempts to transform it to other forms of expressions defined by the formal syntax, such as the previously mentioned expression `e1,e2,e3`. Unlike Compatible Substructure Hoisting and Quantified Node Reduction, converting a structure to another form does not ensure the resulted structure is smaller than the original one, and the number of structures that can be possibly generated is vast or even infinite. To tackle this challenge, we establish five guiding principles for SFC based on the characteristics of program reduction, in order to constrain the space of generated programs. Additionally, we design three reduction methods—Smaller Structure Replacement, Identifier Elimination, and Structure Canonicalization—to effectively apply SFC in program reduction. Specifically, Smaller Structure Replacement reduces a program by replacing structures with smaller, compatible ones generated by SFC. Identifier Elimination strives to create reduction opportunities by utilizing SFC to eliminate the uses of an identifier, in the hope of successfully removing the corresponding definition or initialization of the identifier. While these two methods focus on enhancing minimization, Structure Canonicalization emphasizes canonicalization. It transforms structures into the most canonical forms with SFC, making duplicate programs (*i.e.*, those that trigger

the same bug) more likely to be reduced to identical or similar forms. By integrating these three reduction methods into existing language-agnostic program reducers, *Perses* and *Vulcan* [221], we develop two prototype reducers: $\text{SFC}_{\text{Perses}}$ and $\text{SFC}_{\text{Vulcan}}$.

To evaluate the performance of SFC in minimization and canonicalization, we utilize two benchmark suites, **Benchmark-Reduce** and **Benchmark-Cano**, respectively. **Benchmark-Reduce** contains 20 C benchmarks from *Perses* [192], 20 Rust benchmarks partially from *Vulcan* [221], and 205 SMT-LIBv2 benchmarks from ddSMT2.0 [114]. Each benchmark consists of a program triggering a unique real-world crash or miscompilation bug. In contrast, **Benchmark-Cano** contains many duplicated C programs that trigger the same bug, including 2,501 programs triggering 11 unique crash bugs in GCC 4.3.0, and 1,295 programs triggering 35 miscompilation bugs also in GCC 4.3.0 [45]. Experiments with **Benchmark-Reduce** show that the results of $\text{SFC}_{\text{Perses}}$ is 36.82%, 18.71%, and 41.05% smaller than those of *Perses* on the C, Rust, and SMT-LIBv2 benchmarks, respectively. In terms of execution time, $\text{SFC}_{\text{Perses}}$ takes $3.65\times$, $16.99\times$, and $3.97\times$ the time of *Perses* on average. As for $\text{SFC}_{\text{Vulcan}}$, it can produce 14.51%, 7.65%, and 7.66% smaller results than *Vulcan* at the cost of $1.56\times$, $2.35\times$, and $1.42\times$ the execution time of *Vulcan*, respectively. Moreover, experiments with **Benchmark-Cano** show that $\text{SFC}_{\text{Perses}}$ and $\text{SFC}_{\text{Vulcan}}$ can reduce 442 and 435 more duplicates (programs that trigger the same bug) to identical programs than *Perses* and *Vulcan*, respectively.

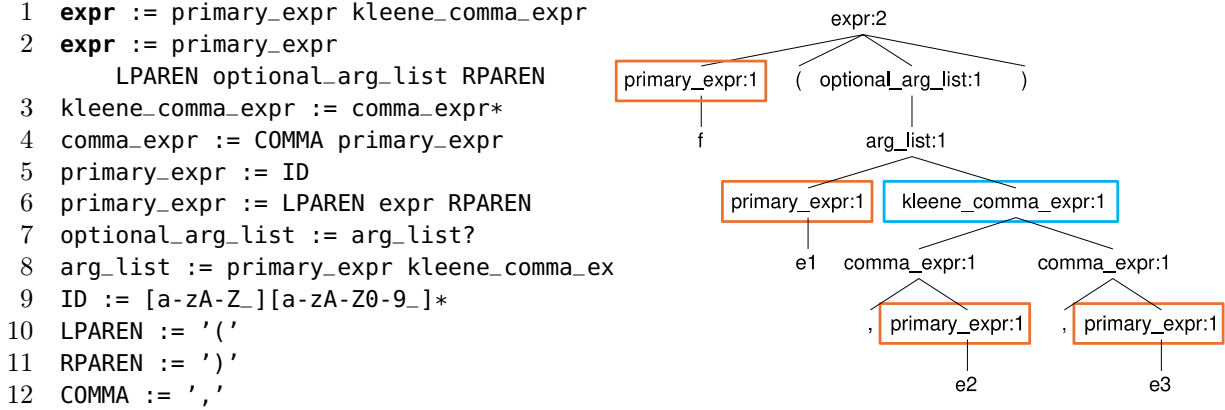
Contributions. This chapter makes the following contributions.

- We propose a novel syntax-guided transformation, Structure Form Conversion, complementing prior transformations Compatible Substructure Hoisting and Quantified Node Reduction.
- We propose three novel reduction methods to effectively and efficiently leverage SFC in program reduction. We further implement two language-agnostic reducer prototypes, $\text{SFC}_{\text{Perses}}$ and $\text{SFC}_{\text{Vulcan}}$ by integrating these methods into existing reducers.
- By conducting extensive evaluation, we demonstrate that $\text{SFC}_{\text{Perses}}$ and $\text{SFC}_{\text{Vulcan}}$ significantly outperform *Perses* and *Vulcan* in terms of both minimization and canonicalization capability.

5.2 Preliminary

This section provides the essential background knowledge relevant to this work.

5.2.1 Syntax of Programming Languages



(a) An example CFG that defines a language expression $f(e_1, e_2, e_3)$ based on the grammar of simple expressions. The bolded **expr** is the shown in Figure 5.1a. For simplicity, in the example, e_1 , e_2 , and e_3 are simply three variables.

Figure 5.1: An example CFG and an example parse tree.

The syntax of a programming language is a set of rules that define the valid textual structure of programs written in this language. Typically, the syntax of a programming language can be formally expressed with a context-free grammar (CFG). A CFG G consists of four components: a finite set of nonterminal symbols N , a finite set of terminal symbols Σ , a finite set of production (or rewrite) rules R , and a start symbol $S \in N$, *i.e.*, $G = (N, \Sigma, R, S)$. The language defined by a CFG G is called a context-free language (CFL), denoted as $L(G)$. Figure 5.1a shows an example CFG that formally defines a language of simple expressions, and Figure 5.1b shows the parse tree of an example expression, $f(e_1, e_2, e_3)$, which conforms to the example CFG.

Nonterminal Symbol. A nonterminal symbol $v \in N$ represents a syntactic structure in the language defined by the corresponding CFG. Each non-leaf sub-tree st (a sub-tree whose height is not 0, also referred to as a structure in this chapter) in the parse tree of a program corresponds to a nonterminal symbol v in the grammar of the language, denoted as $st \mapsto v$ in this chapter. For example, in Figure 5.1b, sub-trees rooted at nodes marked with an orange box correspond to the nonterminal symbol **primary_expr**. In Figure 5.1a, nonterminal symbols are all named with lowercase words while terminal

symbols are uppercased.

Terminal Symbol. While nonterminal symbols can be rewritten to other symbol(s), a terminal symbol $\sigma \in \Sigma$ directly maps to character(s) (a token) that appear in the actual textual content and cannot be expanded to other symbols. Terminal symbols are either defined with constant character(s) or regular expressions. They form the alphabet of the language defined by the grammar.

Production Rule. A production rule $r \in R$ defines how a nonterminal symbol can be rewritten. Its left-hand side is the nonterminal symbol to be defined and the right-hand side is a sequence of symbols (either terminal or nonterminal symbols). Note that a nonterminal can have multiple different production rules. In the example grammar, there are eight production rules (line 1-8), and the nonterminal `expr` has two different production rules (*i.e.*, $|R_{\text{expr}}| = 2$), which means it can be rewritten in two different ways. Given a non-leaf sub-tree st and the nonterminal v that st corresponds to, we can map st to a production rule $r \in R_v$ (denoted as $st \mapsto r^2$), indicating that production rule r is applied to the root of st . In Figure 5.1b, it can be observed that each non-leaf node is written in the format of `nonterminal:index`. The `index` after the colon is the index of the production rule to which the sub-tree corresponds. For example, the root of the entire parse tree is `expr:2`, indicating the parse tree corresponds to the second production rule of `expr`.

Start Symbol. The start symbol of the CFG is a special nonterminal symbol. All the strings that can be represented by the start symbol (*i.e.*, can be possibly derived by applying a sequence of production rules starting from the start symbol) form the language defined by the grammar. In the example grammar, `expr` is defined to be the start symbol.

Normal Forms of CFGs. The CFG can be written in various ways to represent a certain CFL. To impose specific restrictions thus facilitating certain parsing algorithms or theoretical analysis on top of the grammar, many normal forms of CFGs are proposed, such as Chomsky Normal Form (CNF) [48] and Greibach Normal Form (GNF) [86]. In this work, we make the CFG conform to Perses Normal Form (PNF), which is proposed and utilized by the previously proposed syntax-guided program reducer, Perses [192]. PNF requires that each production rule is in one of the following forms: (1) $A ::= B_1 B_2 \dots B_n$ (2) $A ::= B_1^*$ (3) $A ::= B_1^+$ (4) $A ::= B_1^?$, and (5) $S ::= \epsilon$. Symbol A represents any nonterminal symbol, B_i ($\forall i \in [1, n]$) represents any nonterminal symbol or terminal symbol, and S represents the start symbol. The example grammar in Figure 5.1a conforms to PNF.

²We overload the symbol $\mapsto$ to denote both types of mappings (*i.e.*, mappings from a sub-tree to a nonterminal symbol and from a sub-tree to a production rule)

5.2.2 Syntax-Guided Transformation

Syntax-guided transformations are widely used in the process of program reduction to improve its effectiveness and efficiency [192, 170]. The insight behind this is that invalid programs are unlikely to trigger the bug that the original program reveals, and avoiding generating invalid programs can significantly shrink the search space of program reduction, thus saving considerable reduction time. The following definition formally defines syntax-guided transformations.

Definition 5.2.1 (Syntax-Guided Transformation). *Assuming $\mathbb{P}_{valid}$ is the set of all syntactically valid programs, a transformation f_t is a syntax-guided transformation if and only if $\forall P \in \mathbb{P}_{valid}, f_t(P) \in \mathbb{P}_{valid}$.*

Two syntax-guided transformations are adopted by prior language-agnostic program reducer [192].

Compatible Substructure Hoisting: Given a tree, this transformation attempts to simplify the tree by hoisting one of its compatible descendants. A descendant is considered to be compatible if the nonterminal that the root corresponds to can be directly or indirectly rewritten by the nonterminal of the descendant. In Figure 5.1b, the entire parse tree represents $f(e1, e2, e3)$ and corresponds to the nonterminal `expr`. Among all its descendants, four orange nodes corresponding to `primary_expr` are compatible, since according to line 1 in Figure 5.1a, `expr` can be rewritten by `primary_expr` (`kleene_comma_expr` can be empty). Thus, eventually, Compatible Substructure Hoisting attempts to simplify the original parse tree by reducing it to each of four compatible descendants (`f`, `e1`, `e2`, and `e3`).

Quantified Node Reduction: This transformation aims to simplify trees that correspond to a production rule in the form of $A ::= B_1^*$ or $A ::= B_1^+$, or $A ::= B_1^?$. For example, the sub-tree rooted at the blue node in Figure 5.1b corresponds to the production rule `kleene_comma_expr ::= comma_expr*` which is in the form of $A ::= B_1^*$. For such a sub-tree, all its children can be deleted without breaking the syntax; thus Quantified Node Reduction attempts to simplify the tree by applying `ddmin` to its child list. For trees corresponding to production rules like $A ::= B_1^+$, Quantified Node Reduction additionally ensures that at least one child remains, and for trees corresponding to $A ::= B_1^?$, Quantified Node Reduction simply attempts to remove the single child.

5.3 Methodology

This section introduces Structure Form Conversion (SFC) (§ 5.3.1), how to effectively and efficiently apply SFC to the program reduction task (§ 5.3.2), and the implementation (§ 5.3.3)

5.3.1 Structure Form Conversion (SFC)

SFC converts a structure to another form. For example, given a function invocation expression $\mathbf{f}(\mathbf{e1}, \mathbf{e2}, \mathbf{e3})$ in a C program, SFC may convert it to $\mathbf{e1}, \mathbf{e2}, \mathbf{e3}$. Such a conversion complements Compatible Substructure Hoisting and Quantified Node Reduction and may create extra reduction opportunities.

Algorithm 9 describes the general idea of SFC. Given the formal syntax of the programming language G^3 and a sub-tree st in the parse tree of the program written in the specific language, SFC can produce a list of transformed sub-trees T that correspond to the same nonterminal as the original sub-tree st . Specifically, SFC first obtains the nonterminal v that the subtree st corresponds to, *i.e.*, $st \mapsto v$ (line 1). Next, SFC retrieves all the production rules of the nonterminal v in the grammar G (line 2). Then, for each production rule, SFC builds a list of new sub-trees (line 3-6), and eventually all the sub-trees built based on the production rules are returned (line 7).

Algorithm 9: The SFC Algorithm – $\text{SFC}(st, G, \text{pred})$

Input : st , the sub-tree to be transformed.
 G , the formal syntax of the programming language.
 pred , an additional predicate for matching compatible sub-trees

Output : A list of transformed sub-trees T

```

1  $T \leftarrow []$  ;  $v \leftarrow$  the nonterminal symbol that  $st$  maps to, i.e.,  $st \mapsto v$ 
2  $R_v \leftarrow$  the production rules for  $v$ 
3 for  $i \in [0, R_v.\text{size}() - 1]$  do
4   if  $st \mapsto R_v[i]$  then continue           // skip the production rule that the original
   sub-tree corresponds to
5    $TR_v[i] \leftarrow \text{buildSubTreesForTheRule}(R_v[i], st, \text{pred})$ 
6    $T.\text{addAll}(TR_v[i])$ 
7 return  $T$ 
```

³In the algorithm, it is assumed that the grammar is in Perses Normal Form (PNF).

Design Principles

The most essential and challenging part of SFC is the algorithm for building sub-trees based on a given production rule. Making this algorithm effective and efficient for program reduction faces two main challenges. First, given a production rule, there is often an infinite (or at least extremely large) number of valid sub-trees that can be generated. Therefore, the algorithm needs to be properly guided and restricted. Otherwise, it will be too expensive to be feasible. Second, randomly generated sub-trees may render the semantics of the program invalid even though they preserve the syntactic validity. Although previous approaches like Perses face the same problem as they cannot ensure semantic validity during the reduction, randomly generating sub-trees can amplify this problem and thus significantly impair the reduction performance. For example, a randomly generated expression is very likely to contain a random identifier that is not defined in the program. In such a case, the program after the replacement is not likely to trigger the same bug, as the language processor probably terminates when it detects this semantic error in an early stage.

To overcome the aforementioned two challenges, we propose five principles to restrict the sub-tree building algorithm. The insight behind all the principles is that the fewer changes are made, the more likely the bug-triggering property can be preserved. In other words, instead of aggressively simplifying the problem in a single step, gradually reducing it through small steps is more likely to succeed, especially since SFC is designed to operate on programs that have already been reduced to a relatively small size. Therefore, the essence of these principles is to avoid making excessive and unnecessary changes to the original sub-trees.

Principle 1: *Do not randomly generate any structure. Reuse the structures in the original sub-tree instead.* This principle is to avoid generating numerous sub-trees that are unlikely to replace the original sub-tree without making the program no longer trigger the same bug. For example, given a sub-tree that represents $f(e1, e2, e3)$, generating a new expression with some random identifiers is usually meaningless as such an expression would probably lead to an undefined identifier error.

Principle 2: *A structure in the original sub-tree can only be reused once.* This principle further reduces the number of sub-trees that can be possibly generated. Meanwhile, it can also reduce the difference between the generated sub-tree and the original sub-tree, thus increasing the possibility that replacing the original sub-tree with the new one makes the program still trigger the same bug.

Principle 3: *The reused structures should remain in the same order as they appear in the original sub-tree.* Similar to principle 2, this principle narrows down the search space of the algorithm and makes the generated sub-trees be more similar to the original sub-tree.

Principle 4: *The reused structure should be as high-level as possible.* This principle aims to avoid generating sub-trees that are aggressively simplified. Meanwhile, adhering to this principle makes SFC more orthogonal to Compatible Substructure Hoisting. It makes SFC focus on converting the structure to another form and delegates the simplification task to the subsequent Compatible Substructure Hoisting.

Principle 5: *Reuse as many structures as possible.* When the input production rule is in the form of $A ::= B_1*$ or $A ::= B_1+$, the algorithm needs to decide the number of structures corresponding to B_1 that the generated sub-trees should contain. This principle guides the algorithm to maximize this number while not breaking other principles. Similar to principle 4, this principle can avoid generating aggressively simplified sub-trees. Meanwhile, it makes SFC more orthogonal to Quantified Node Reduction.

Sub-Tree Building

Algorithm 10 describes the sub-tree building algorithm of SFC. This algorithm takes three inputs, *i.e.*, a sub-tree to be transformed st , a production rule r , and an additional predicate for matching sub-trees (this input is for identifier elimination and will be discussed in § 5.3.2). The output of the algorithm is a list of sub-trees that are generated based on the input production rule r . Recall that in PNF, the right hand side of a production rule can be a sequence of symbols or a single symbol followed by a kleene star (*), or a kleene plus (+), or an optional quantifier (?).⁴ For different forms of production rules, the ways to build sub-trees are different.

Building for Production Rules with a Quantified Symbol. When the right hand side of the production rule r is a single quantified symbol, the algorithm first extracts the single symbol s (line 3), and then searches for all the compatible sub-trees in st , *i.e.*, the tree to be transformed (line 4). If no compatible st is found, meaning no sub-tree can be reused for the building, the algorithm simply returns an empty list, indicating no sub-tree can be built (line 5). Otherwise, the algorithm builds sub-trees according to the type of quantifier that follows the symbol s . If the symbol s is followed by an optional quantifier (?), the algorithm traverses the found compatible sub-trees. For each compatible sub-tree, if it is at the highest level⁵ (line 8), a transformed sub-tree is built by simply reusing the compatible sub-tree (line 6-9). If the symbol s is followed by a Kleene star (*) or Kleene

⁴The special form $S ::= \epsilon$ for including empty string in the defined language is omitted here.

⁵A compatible node is at the highest level if none of its ancestors is compatible. According to the fourth aforementioned design principle, the algorithm is designed to prefer reusing node as high-level as possible.

Algorithm 10: The Sub-Tree Building Algorithm – `buildSubTreesForTheRule( $r, st, \text{pred}$ )`

Input : st , the sub-tree to be transformed.
 r , the production rule based on which the sub-tree is built.
 pred , an additional predicate for matching compatible sub-trees

Output : A list of built sub-trees T

```

1   $T \leftarrow []$ 
2  if The form of the production rule  $r$  is  $A ::= B_1*$  or  $A ::= B_1+$  or  $A ::= B_1?$  then
3       $s \leftarrow$  the symbol in the right hand side of the production rule
4       $\text{compatibleTrees} \leftarrow \text{findCompatibleSubTrees}(s, st, \text{pred})$ 
5      if  $\text{compatibleTrees.size()} == 0$  then return  $[]$ 
6      if  $s$  is followed by an optional quantifier (?) then
7          foreach  $ct \in \text{compatibleTrees}$  do
8              if  $\exists ct' \in \text{compatibleTrees}$ , s.t.  $ct$  is a descendant of  $ct'$  then continue
9                  // Principle 4
9               $t' \leftarrow$  a subtree built based on  $r$  by reusing  $ct$ ;  $T.\text{append}(t')$ 
10     else //  $s$  is followed by a Kleene Star (*) or a Kleene Plus (+)
11          $t' \leftarrow$  a subtree built based on  $r$  by reusing all highest-level compatible sub-trees in
12              $\text{compatibleTrees}$ 
12          $T.\text{append}(t')$ 
13 else // The form of the production rule  $r$  is  $A ::= B_1B_2...B_n$ 
14      $\text{symbolList} \leftarrow$  all the symbols in the right hand side of the production rule  $r$ 
15      $\text{uniqueSymbols} \leftarrow \text{symbolList.toSet}()$ 
16      $\text{symbolToCompatibleMap} \leftarrow$  an empty map
17     foreach  $s$  in  $\text{uniqueSymbols}$  do
18         if  $s$  is a terminal and is defined with fixed text then continue // No need to match
19             for constants
19         else
20              $\text{compatibleTrees} \leftarrow \text{findCompatibleSubTrees}(s, st, \text{pred})$ 
21              $n \leftarrow$  the number of symbol  $s$  in  $\text{symbolList}$ 
22             if  $n > \text{compatibleTrees.size}()$  then return  $[]$ 
23              $\text{symbolToCompatibleMap}[s] \leftarrow$  all  $n$ -combinations of  $\text{compatibleTrees}$ 
24             //  $\text{symbolToCompatibleMap}$  now records all combinations of compatible subtrees
25             for each unique symbol
26              $\text{reusings} \leftarrow$  the cartesian product of  $\text{symbolToCompatibleMap}[s]$  of each unique symbol  $s$ 
27              $\text{adoptedReusings} \leftarrow$  all elements of  $\text{reusings}$  in which all subtrees are independent and as
28                 high-level as possible // Adhere to principle 2 and 4
26             foreach  $\text{reusing}$  in  $\text{adoptedReusings}$  do
27                  $t' \leftarrow$  a subtree built based on  $r$  by reusing subtrees in  $\text{reusing}$ ;  $T.\text{append}(t')$ 
28 return  $T$ 

```

plus (+), the algorithm only builds one transformed sub-tree by reusing all the highest-level compatible sub-trees⁵ (line 10-12).

Building for Production Rules with a Sequence of Symbols. If the production rule r is in the form of $A ::= B_1 B_2 \dots B_n$, the algorithm needs to find a compatible sub-tree for each symbol in the right hand side of r (except terminal symbols with constant text such as **COMMA** and **LPAREN**) to reuse. In other words, a sequence of compatible sub-trees in which all the symbols in the right hand side of r can have a compatible sub-tree to reuse (such a sequence is referred to as a matching for brevity) is required to build one transformed sub-tree. To illustrate the algorithm in this scenario, we describe the steps with a running example. We use the parse tree shown in Figure 5.1b as the input of the running example. Let the input production rule be **expr** := **primary_expr** **kleene_comma_expr** (line 1 in Figure 5.1a), and suppose there is no additional predicate.

Initialization: First, the algorithm initializes **symbolList** with all the symbols on the right hand side of the production rule r (line 14), and extracts all the unique symbols to a set **uniqueSymbols** (line 15). In the running example, both **symbolList** and **uniqueSymbols** contain **primary_expr** and **kleene_comma_expr**. It should be noted that in practice, a symbol can appear multiple times in the right hand side of a production rule. Next, **symbolToCompatibleMap**, a map that maps each unique symbol to all the possible matchings, is initialized with an empty map (line 16).

Searching for Compatible Sub-Trees: The algorithm traverses **uniqueSymbols**. For each unique symbol s , if it is a terminal symbol and is defined with constant text, then no compatible sub-tree is needed and the algorithm moves to the next unique symbol (line 18). Otherwise, the algorithm searches for all the compatible sub-trees for the symbol s (line 20). In the running example, the symbol **primary_expr** has four compatible sub-trees, whose roots are marked with orange and the symbol **kleene_comma_expr** has one compatible sub-tree, whose root is marked with blue.

Obtaining Matchings for Each Unique Symbol: Next, the number of appearances n of this unique symbol is retrieved (line 21). Recall that to build a transformed sub-tree, each symbol needs a compatible sub-tree to reuse. Therefore, the number n is also the required number of sub-trees to form a matching for the unique symbol. If the total number of found compatible sub-trees is smaller than n , the algorithm returns an empty list, meaning no transformed sub-tree can be generated (line 22). Otherwise, all n -combinations of the found compatible trees are stored as matchings in **symbolToCompatibleMap** with the corresponding unique symbol s as the key. In the running example, both unique symbols appear only once in the production rule. Thus, the number n for both unique symbols equals one, and the matchings for these two unique symbols are the 1-combinations of their compatible

sub-trees. Specifically, there are four matchings for the unique symbol `primary_expr` and one matching for `kleene_comma_expr`.

Obtaining Matchings for the Entire Production Rule: Once the matchings of each unique symbol is found, the algorithm derives all the matchings for the entire right hand side of r by calculating the Cartesian product of the matchings of each unique symbol (line 24). In the running example, `primary_expr` has four matchings. Each of them contains one sub-tree that represents `'f'`, `'e1'`, `'e2'`, `'e3'`, respectively. `kleene_comma_expr` has one matching containing one sub-tree representing `'e2,e3'`. Therefore the resulted Cartesian product contains four matchings representing `'f,e2,e3'`, `'e1,e2,e3'`, `'e2,e2,e3'`, `'e3,e2,e3'`. However, the last two matchings violate the second design principle (they reuse `'e2'` and `'e3'` twice, respectively), the algorithm filters such matchings by excluding all matchings that contain dependent compatible sub-trees (line 25). Finally, all the remaining matchings are utilized to build transformed sub-trees (line 26-27).

5.3.2 Program Reduction with SFC

Unlike Compatible Substructure Hoisting and Quantified Node Reduction, SFC does not ensure that the resulting structure is smaller than the original structure. Meanwhile, compared to the previous two syntax-guided transformations, the number of structures that can be possibly generated by SFC is usually much larger. Therefore, to effectively and efficiently adapt SFC to the program reduction task, we further propose three reduction methods: Smaller Structure Replacement, Identifier Elimination via SFC and Structure Canonicalization. This section details how these reduction methods utilize SFC to boost program reduction.

Smaller Structure Replacement

This reduction method aims to reduce a program by replacing structures of the program with their smaller forms. It traverses the structures at different levels in the parse tree of the program, invokes the SFC algorithm to generate structures in different forms that are smaller, and attempts to replace the original structure with the newly generated structure. If the program after the replacement still triggers the same bug, the program is successfully reduced.

Algorithm 11 describes this reduction method. Given a bug-triggering program P to be reduced, the formal syntax of the language in which P is written, and an oracle that can check whether a program triggers the specific bug, the algorithm can produce a reduced

Algorithm 11: Smaller Structure Replacement– `reduceBySimplifyingStructure( $t, G$ )`

Input : P , the program to be reduced.
 G , the formal syntax of the programming language.
 ψ , the oracle that verifies whether a program triggers the specific bug

Output : The reduced program p

```
1  $t \leftarrow$  the parse tree of  $P$ ; truePred  $\leftarrow$  a predicate that always returns true
2 queue  $\leftarrow$  a queue that only contains the root node of  $t$ 
3 while queue is not empty do
4   root  $\leftarrow$  queue.poll();  $st \leftarrow$  the sub-tree rooted at root
5    $T \leftarrow$  the result of SFC( $st, G, \text{truePred}$ ) sorted in ascending order
6   for  $st' \in T$  do
7     if the number of leaves in  $st'$  is not smaller than that of  $st$  then break
8      $t' \leftarrow$  a new parse tree obtained by replacing  $st$  with  $st'$ 
9      $p \leftarrow$  the program derived from  $t'$ 
10    if  $\psi(p)$  then  $t \leftarrow t'$ ; root  $\leftarrow$  the root of  $st'$ ; break
11  queue.addAll(root.children)
12 return the program derived by  $t$ 
```

program p by simplifying structures in the program in a breadth-first manner. Specifically, at the beginning, the algorithm obtains the parse tree of the input program P , initializes a predicate that always returns **true** (required by **SFC**, meaning no additional criterion for compatible sub-trees), and adds the root of the parse tree to a queue (line 1-2). Next, the algorithm starts simplifying structures in a while loop (line 3-11). First, the node at the front of the queue and the sub-tree rooted at this node are retrieved (line 4). Next, the algorithm invokes **SFC** to obtain a list of transformed sub-trees and sorts the sub-tree list based on the number of leaves in ascending order (line 5). Then, in a for loop (line 6-10), the algorithm attempts to replace the original sub-tree st with each transformed sub-tree st' until the transformed sub-tree st' is not smaller than the original sub-tree st (line 7) or the program after replacement still triggers the bug (line 10), which means a structure is successfully simplified. After the simplification performed in the for loop, the children of the original sub-tree st (or the transformed sub-tree st' if the simplification succeeded) are added to the queue (line 11), and the algorithm repeats the process to keep simplifying structures in the program.

Algorithm 12: Identifier Elimination– `reduceByEliminatingIdentifiers( $t, G$ )`

Input : P : the program to be reduced.
 G : the formal syntax of the programming language.
 ψ : the oracle that verifies whether a program triggers the specific bug

Output : The transformed program p

```
1 allIdentifiers  $\leftarrow$  all identifier tokens in program  $P$ 
2 nameToldTokensMap  $\leftarrow$  a map that maps each unique identifier name to the corresponding
  identifier tokens
3 foreach name  $\in$  nameToldTokensMap.keys() do
4   | uselds  $\leftarrow$  all the uses of name (i.e., identifier tokens that are not for ID definition or
  | initialization)
5   | foreach anotherName  $\in$  nameToldTokensMap.keys() \ name do
6   |   |  $p \leftarrow$  a new program obtained by renaming all uselds with anotherName
7   |   | if  $\psi(p)$  then return  $p$ 
8   | foreach use  $\in$  uselds do // Eliminate each use by either SFC or replacing
9   |   | foreach anotherName  $\in$  nameToldTokensMap.keys() \ name do
10  |   |   |  $p \leftarrow$  a new program obtained by renaming use with anotherName
11  |   |   | if  $\psi(p)$  then update  $P$  with  $p$  and go to line 8 to eliminate the next use
12  |   | success  $\leftarrow$  eliminateBySFC(use,  $P$ ,  $\psi$ )
13  |   | if  $\neg$ success then go to line 3 and move to next unique identifier name // Fail to
  |   | eliminate
14 return  $P$ 
15 Function eliminateBySFC(use,  $P$ ,  $\psi$ ):
16   | parent  $\leftarrow$  use; pred  $\leftarrow$  a predicate that takes a sub-tree as input and returns true only if the
  |   sub-tree does not contain use
17   | while parent has a parent node do
18   |   | parent  $\leftarrow$  the parent node of parent
19   |   |  $T \leftarrow$  the result of SFC( $st$ ,  $G$ , pred) sorted in ascending order
20   |   | for  $st' \in T$  do
21   |   |   | if the number of leaves in  $st'$  is not smaller than that of  $st$  then break
22   |   |   |  $p \leftarrow$  a program obtained by replacing the sub-tree rooted at parent with  $st'$ 
23   |   |   | if  $\psi(p)$  then update  $P$  with  $p$ ; return true
24   | return false
```

Identifier Elimination

In previous research work, eliminating the uses of an identifier has been demonstrated to be effective for further simplifying a bug-triggering program, since it can potentially make some definitions or initializations of identifiers redundant, and thus deletable [221]. However, previous approaches of identifier elimination rely only on replacements (*i.e.*, replacing all the uses of an identifier with another identifier). The reduction method proposed in this section aims at additionally utilizing SFC to eliminate uses of an identifier for better effectiveness. Specifically, given a use of an identifier, this approach replaces a structure that contains the use with another structure generated by SFC. During the generation, any sub-tree that contains the given use is forbidden to be reused; thus, after the replacement, the use of the identifier is eliminated.

Algorithm 12 describes the complete reduction method in detail. First, given a program to be reduced P , all the identifier tokens in P are collected and clustered by their names (line 1-2). To achieve this, the terminal symbol (*e.g.*, **ID** in Figure 5.1a) that represents identifier tokens in the corresponding grammar G must be specified. Although this requirement introduces some degree of language specificity, we believe it does not compromise the generality of our approach, as it can typically be addressed in a straightforward manner. Next, the algorithm traverses each unique identifier and attempts to eliminate all the uses for the specific identifier (line 20-23). To eliminate all the uses, the algorithm first attempts to rename all of them at once with another unique identifier (line 5-7). If the elimination is successful, the program after elimination is returned for further reduction. Otherwise, the algorithm attempts to eliminate the uses one by one (line 8-13). To do so, the algorithm traverses all the uses, and for each use, it first attempts to rename it in a similar way as renaming all the uses at once (line 9-11). If the attempt fails, the function **eliminateBySFC** is invoked (line 19). This function takes as input a use (*i.e.*, an identifier token), the program P , and the oracle ψ , and then it attempts to eliminate the use by replacing the sub-tree that contains the use with a new sub-tree without the use generated by SFC. Specifically, the function first initializes a predicate **pred** which takes as input a sub-tree and only return **true** when the sub-tree does not contain the input use (line 16). Next, in a while loop, starting from the parent of the leaf node representing the use, the function traverses the ancestors of the use from bottom to the top (line 17-23). For the sub-tree rooted at each ancestor node, a list of sub-trees are generated by invoking SFC (line 19). It should be noted that the predicate **pred** ensures that the generated sub-trees do not contain the use. Finally, for each generated sub-tree st' that is smaller than the original sub-tree st , the algorithm attempts to replace st with st' (line 20-22). If the replacing is successful (*i.e.*, the new program p still triggers the same bug), P is updated

by p and the function immediately returns **true**, indicating the elimination is successful (line 23).

Structure Canonicalization

In addition to minimizing bug-triggering inputs, another desired feature of program reduction is canonicalizing inputs that trigger the same bug. This canonicalization feature can significantly facilitate bug deduplication and help solve the fuzzer taming problem [45], thus being highly demanded in practice. To boost the canonicalization aspect of language-agnostic program reducers, we propose a method named Structure Canonicalization. Its workflow is similar to Smaller Structure Replacement in Algorithm 11. While Smaller Structure Replacement attempts to replace the original structure with a smaller one, Structure Canonicalization attempts to replace the original structure with a structure having the same size but being more canonical. To decide which structure is more canonical, we follow the definition below.

Definition 5.3.1. *Given a nonterminal v and two sub-trees $st \mapsto v$ and $st' \mapsto v$. The subtree st is more canonical than the subtree st' if and only if $st \mapsto r$, $st' \mapsto r'$, and the index of r is smaller than that of r' in R_v .*

For example, suppose we have two different production rules for the nonterminal **expr**, ordered as follows: ① **expr:=ID+ID**; ② **expr:=ID.ID**. In this case, we consider **a+b** to be more canonical than **a.b**. If the order were reversed, then **a.b** would be considered more canonical. The purpose here is that if both **a+b** and **a.b** can trigger the same bug, we want to canonicalize them to a single, canonical form. In our implementation, the production rules are ordered based on their appearance in the Antlr4 grammar file.

The algorithm of Structure Canonicalization can be obtained by adjusting the smaller structure replacement algorithm (Algorithm 11) and the SFC algorithm (Algorithm 9) it invokes. First, **continue** on line 4 in SFC needs to be revised to **break**, so that SFC generates only more canonical sub-trees. Next, line 5 in Algorithm 11 should invoke the revised SFC instead and filter out sub-trees having different size from the original sub-tree st . Finally, line 7 in Algorithm 11 needs to be removed.

5.3.3 Implementation

We implemented two prototypes, $\text{SFC}_{\text{Perses}}$ and $\text{SFC}_{\text{Vulcan}}$ on top of two previous language-agnostic program reducers, Perses and Vulcan, respectively. Figure 5.2 shows the workflow of $\text{SFC}_{\text{Perses}}$, which contains a main reducer Perses and three auxiliary reducers. The

main reducer, *Perses*, is responsible for reducing the input efficiently. The three auxiliary reducers are implemented based on the proposed three SFC-based reduction method and they are supposed to either aggressively search for reduction opportunities and/or create reduction opportunities for the main reducer. The architecture of $\text{SFC}_{\text{Vulcan}}$ is similar to $\text{SFC}_{\text{Perses}}$ but uses *Vulcan* as main reducer. Compared to $\text{SFC}_{\text{Perses}}$, $\text{SFC}_{\text{Vulcan}}$ additionally incorporates the auxiliary reducers proposed by *Vulcan*.

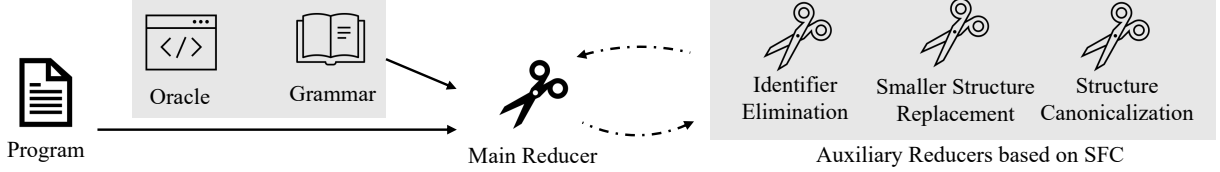


Figure 5.2: The overall workflow of $\text{SFC}_{\text{Perses}}$

Both $\text{SFC}_{\text{Perses}}$ and $\text{SFC}_{\text{Vulcan}}$ take three inputs, *i.e.*, a bug-triggering program, the grammar in which the program is written, and an oracle that verifies whether a program triggers the specific bug. The input program is first reduced by the main reducer. Once the main reducer cannot make any progress, the first auxiliary reducer is invoked to further reduce the program and/or create reduction opportunities by performing certain transformations. If the auxiliary makes progress, the program is sent back to the main reducer, otherwise, the next auxiliary reducer is invoked. For example, if Identifier Elimination successfully eliminates an identifier, the program will be directly sent back to the main reducer. Otherwise, if Identifier Elimination cannot eliminate any identifier, the program will be sent to the next auxiliary reducer, *i.e.*, Smaller Structure Replacement. This alternating process is repeated until the last auxiliary reducer cannot make any progress, and it retains the same theoretical guarantee (e.g., 1-minimality) as the main reducer.

5.4 Evaluation

To evaluate the proposed SFC-based reduction methods, we conducted experiments to investigate the following research questions.

RQ1: Can SFC benefit previous language-agnostic program reducers in minimization?

RQ2: Can SFC benefit previous language-agnostic program reducers in canonicalization?

RQ3: To what extent does each SFC-based method contribute to the performance improvement?

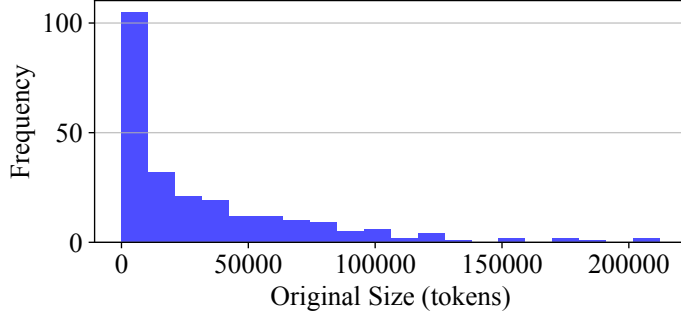


Figure 5.3: Distribution of the original program sizes in **Benchmark-Reduce**.

Benchmark Suites. We use two benchmark suites, **Benchmark-Reduce** and **Benchmark-Cano**, to assess the performance in minimization and canonicalization for each reducer. Specifically, **Benchmark-Reduce** contains 20 C benchmarks, 20 Rust benchmarks, and 205 SMT-LIBv2 benchmarks. Each benchmark consists of a program that triggers a unique real-world crash or miscompilation bug and a shell script that checks whether a program triggers the specific bug. The original sizes of these bug-triggering programs range from 15 to 212,259 tokens, and the distribution of the original sizes is shown in Figure 5.3. We utilize this benchmark suite for assessing the minimization performance of reducers. These benchmarks are also widely used for evaluation in previous studies in this direction [192, 221, 230, 200, 228, 79, 114, 220]. **Benchmark-Cano** contains 3,796 bug-triggering C programs. 2,501 of them trigger 11 unique crash bugs in GCC 4.3.0, and the remaining 1,295 programs trigger 35 miscompilation bugs also in GCC 4.3.0. This benchmark suite is used for evaluating bug deduplication performance in previous studies [45, 222]. In our evaluation, we use this suite mainly for evaluating the canonicalization performance of reducers.

Baselines. To evaluate SFC_{Perses} and SFC_{Vulcan} , we compare them against the corresponding previous language-agnostic program reduction tools, Perses [192] and Vulcan [221], and two program reducers that are specifically optimized for C/C++ and SMT-LIBv2, *i.e.*, C-Reduce [171] and ddSMT [114].

- *Perses* is the first syntax-guided language-agnostic program reducer. It is implemented with two syntax-guided transformations, Compatible Substructure Hoisting and Quantified Node Reduction, to achieve efficient program reduction.
- *Vulcan* is a language-agnostic program reduction framework that aims to trade off the

efficiency for a smaller reduction result. It deploys Perses as the main reducer and includes three auxiliary reducers to search for reduction opportunities when Perses cannot make any progress.

- *T-Rec* is a fine-grained language-agnostic program reduction technique that mainly aims to improve canonicalization by searching for reduction opportunities within tokens [220]. We included it as a baseline when evaluating the canonicalization performance. Unlike Perses, Vulcan, and SFC—which treat tokens as atomic elements—T-Rec is capable of reducing and canonicalizing token content. For example, it can reduce a random string literal to an empty string or a numeric constant to 0. We chose to include T-Rec in our evaluation of canonicalization for two reasons. First, T-Rec also aims to improve canonicalization. By integrating it, we demonstrate that SFC is complementary to T-Rec and that combining them can further enhance canonicalization performance. Second, integrating T-Rec simplifies the evaluation. Although SFC improves canonicalization by aligning program structure, few duplicates are reduced to exactly identical programs because Perses, Vulcan, and SFC do not modify characters within token contents. As a result, structurally similar programs may still differ in unimportant literals. T-Rec addresses this by canonicalizing tokens, thereby enabling more duplicates to be textually identical, which simplifies downstream deduplication tasks. T-Rec is implemented as an additional auxiliary reducer, integrated by appending it to the list of auxiliary reducers.
- *C-Reduce* and *ddSMT* are program reducers specifically optimized for reducing C/C++ and SMT-LIBv2 programs, respectively. By comparing our approach with C-Reduce and ddSMT, we aim to assess the extent to which SFC narrows the gap between language-agnostic and language-specific reducers. Since C-Reduce also includes certain language-agnostic reduction components, it can be applied to programs in other languages as well. Therefore, we include C-Reduce as a baseline for the Rust benchmarks. However, we exclude C-Reduce when evaluating on SMT-LIBv2 benchmarks, as we observed that it often fails to reduce these programs effectively within a reasonable time.

Metrics. We use two metrics in our evaluation. To assess the minimization capability of a reducer, we define the **Size** of a program by the number of tokens it contains. A smaller output indicates a stronger minimization capability. We use **Number of Eliminated Duplicates** to measure the performance of canonicalization. Given a list of duplicates, assume the number of distinct programs that the list contains is n . Suppose that after reduction, the number of distinct programs becomes m . Then the difference (*i.e.*, $n - m$) is the number of eliminated duplicates. A larger difference indicates that more duplicates become identical after reduction (thus can be eliminated), demonstrating a stronger

canonicalization capability.

Experiment Setup. The experiments are run on an Ubuntu 20.04 server with AMD 7950X CPU and 128 GB RAM. For fair comparisons, all the experiments are run with a single thread.

5.4.1 RQ1: Performance in Minimization

To investigate this research question, we compare SFC_{Perses} and SFC_{Vulcan} to Perses and Vulcan on **Benchmark-Reduce**, respectively. Figure 5.4, Figure 5.5, and Figure 5.6 show the results of the experiment. For the C and Rust benchmarks, detailed results are also listed in Table 5.1 and Table 5.2.

SFC_{Perses} vs. Perses. As shown in Table 5.1 and Figure 5.4-5.6, the results produced by SFC_{Perses} are generally and significantly smaller than those of Perses, indicating that integrating SFC-based reduction methods can effectively boost the minimization performance of Perses. Specifically, in terms of size, the results of SFC_{Perses} are 36.82%, 18.71%, and 41.05% smaller than those of Perses on average on C, Rust, and SMT-LIBv2 benchmarks, respectively. In terms of execution time, incorporating SFC-based methods inevitably introduces overhead. On average, SFC_{Perses} takes $3.65\times$, $16.99\times$, and $3.97\times$ the time of Perses on C, Rust, and SMT-LIBv2 benchmarks. The unusually high time ratio on the Rust benchmarks is primarily due to the presence of many small benchmarks, which inflates the average. For instance, when considering only the 10 smallest benchmarks from `rust-77320` to `rust-78720`, the time ratios of these benchmarks range from 11 to 64, averaging 26.46. However, the absolute time differences for these benchmarks range from 82s to 628s, averaging 282.8s—less dramatic than the ratios suggest.

SFC_{Perses} vs. Vulcan. From Table 5.1 and Figure 5.4-5.6, it can be observed that the overall performance of SFC_{Perses} and Vulcan is comparable. Specifically, the results of SFC_{Perses} are 1.86%, 3.34%, and 3.34% smaller than those of Vulcan. The p-values yielded by the Wilcoxon signed-rank test are 0.105, 0.530, and 0.598, respectively, which indicate that there is no statistically significant difference between the two reducers in terms of minimization performance. Time-wise, SFC_{Perses} takes $1.19\times$, $1.33\times$, and $0.95\times$ the time of Vulcan on C, Rust, and SMT-LIBv2 benchmarks, and the corresponding p-values are 0.04, 0.812, and 1.256×10^{-6} , respectively. These results indicate that statistically, SFC_{Perses} is significantly slower than Vulcan on C benchmarks, significantly faster on SMT-LIBv2 benchmarks, and not significantly different on the Rust benchmarks. Although SFC_{Perses} and Vulcan achieve comparable overall minimization performance, it is worth noting that they are complementary to some extent. Specifically, in 53 out of 245 benchmarks, SFC_{Perses}

produces results that are more than 10% smaller than **Vulcan**, while in 48 benchmarks, it produces results that are more than 10% larger. Furthermore, our comparison of **SFC_{Vulcan}** with **Vulcan** demonstrates that integrating SFC-based reduction methods into **Vulcan** leads to further improvements in minimization.

SFC_{Vulcan} vs. Vulcan vs. C-Reduce and ddSMT. As illustrated in Table 5.2 and Figure 5.4-5.6, **SFC_{Vulcan}** significantly outperforms **Vulcan** in terms of minimization. On average, the results of **SFC_{Vulcan}** are 14.51%, 7.65%, and 7.66% smaller than those of **Vulcan** on C, Rust, and SMT-LIBv2 benchmarks, respectively. However, on C benchmarks and SMT-LIBv2 benchmarks, **SFC_{Vulcan}** does not outperform the corresponding language-specific reducers, *i.e.*, C-Reduce and ddSMT, which is expected. Specifically, **SFC_{Vulcan}** produces 50.39% and 7.55% larger results than C-Reduce (on C benchmarks) and ddSMT (on SMT-LIBv2 benchmarks), respectively. In contrast, the results of **Vulcan** are 80.38% and 16.22% larger than those of C-Reduce and ddSMT, showing that **SFC_{Vulcan}** closes a substantial portion of the gap. For the Rust benchmarks, although C-Reduce is also effective at reducing Rust programs, **SFC_{Vulcan}** achieves better results—producing outputs that are, on average, 11.58% smaller than those of C-Reduce. In terms of execution time, **SFC_{Vulcan}** takes 1.56×, 2.35×, and 1.42× the time of **Vulcan** on C, Rust, and SMT-LIBv2 benchmarks. Compared to language-specific reducers, **SFC_{Vulcan}** takes 1.85× and 3.22× the time of C-Reduce on C and Rust benchmarks, and 6.04× the time of ddSMT on SMT-LIBv2 benchmarks.

Trade-off between Time Overhead and Reduction. As shown in the results discussed above, SFC introduces a notable time overhead. However, it should be noted that the reducer always maintains a minimal result during the reduction process, allowing users to stop at any point that they deem the result sufficiently simplified. The objective of SFC is not to replace **Perses** or **Vulcan**, but rather to offer an effective and efficient method for users to further trade execution time for a smaller result when needed. Given that minimizing bug-triggering programs is not always time-sensitive, and manual inspection and minimization can be labor-intensive—demanding specific language and compiler expertise—we believe providing the option to trade additional CPU time for a smaller program to initiate manual analysis is beneficial.

Example Simplifications Performed by SFC. To provide insights on how SFC-based reduction methods enable further reduction, we manually inspect some of the C benchmarks and summarize some example simplifications performed by the SFC-based reduction methods shown below (in the examples, **ei** represents an expression, **si** represents a statement, and **a** and **b** are identifiers):

- | | |
|--|---|
| 1. <code>e1 && e2 ==> !e1</code> | 7. <code>e1.e2 = e3 ==> ++e1.e2</code> |
| 2. <code>e1 < e2 ==> !e1</code> | 8. <code>f(e1, e2) ==> --e1</code> |
| 3. <code>e1.e2 ==> &e1</code> | 9. <code>s1 ==> ;</code> |
| 4. <code>e1.e2 ==> sizeof e1</code> | 10. <code>if (a=b); ==> a=b;</code> |
| 5. <code>e1[e2] ==> *e1</code> | 11. <code>for(;a;a++) ==> while(a++)</code> |
| 6. <code>e1[e2].[e3] ==> e1->e3</code> | 12. <code>for(;;) { s1 s2 } ==> for(s1;;s2)</code> |

RQ1: While SFC-based reduction methods introduce some time overhead, they significantly boost the minimization capabilities of *Perses* and *Vulcan*. Specifically, the results of $\text{SFC}_{\text{Perses}}$ are 36.82%, 18.71%, and 41.05% smaller than those of *Perses* on C, Rust, and SMT-LIBv2 benchmarks. As for $\text{SFC}_{\text{Vulcan}}$, the corresponding percentage decreases are 14.51%, 7.65%, and 7.66%, respectively.

5.4.2 RQ2: Performance in Canonicalization

To evaluate whether SFC-based reduction methods can boost performance in canonicalization, we run *Perses*, *Vulcan*, $\text{SFC}_{\text{Perses}}$ and $\text{SFC}_{\text{Vulcan}}$ to reduce all the bug-triggering programs in **Benchmark-Cano**. As mentioned previously, to demonstrate that SFC-based reduction methods are complementary to T-Rec, another technique for improving canonicalization performance, we integrate T-Rec to each reducer in the experiment.

The evaluation results are shown in Table 5.3 ($\text{SFC}_{\text{Perses}}^{\text{--sc}}$ is a variant for the ablation study covered in § 5.4.3, where its results are discussed). From the table, it can be observed that compared to *Perses*, $\text{SFC}_{\text{Perses}}$ can eliminate 439 more crash duplicates and 3 more miscompilation duplicates. Meanwhile, compared to *Vulcan*, $\text{SFC}_{\text{Vulcan}}$ eliminates 425 and 10 more crash and miscompilation duplicates, respectively. Such results indicate that SFC-based reduction methods significantly improve the canonicalization capabilities of both *Perses* and *Vulcan*. Time-wise, the average execution time of $\text{SFC}_{\text{Perses}}$ is $1.69\times$ and $3.08\times$ the time of *Perses* on the crash and miscompilation benchmarks, respectively. As for $\text{SFC}_{\text{Vulcan}}$, its average time is $1.06\times$ and $1.35\times$ the time of *Vulcan*, respectively.

Table 5.1: Evaluation results of Perses, Vulcan, and $\text{SFC}_{\text{Perses}}$ on C and Rust benchmarks. Column $\% \Delta$ shows the average **percentage change in size** achieved compared to the baseline reducer.

Benchmarks	Perses		Vulcan		$\text{SFC}_{\text{Perses}}$					
	Time (s)	Size (#)	Time (s)	Size (#)	Time (s)	Size (#)	$\% \Delta$ in size <i>w.r.t.</i> Perses	Time ratio <i>w.r.t.</i> Perses	$\% \Delta$ in size <i>w.r.t.</i> Vulcan	Time ratio <i>w.r.t.</i> Vulcan
clang-22382	508	144	1,058	108	1,114	113	-21.53%	2.19	4.63%	1.05
clang-22704	1,376	78	1,574	62	1,650	61	-21.79%	1.20	-1.61%	1.05
clang-23309	1,619	464	6,802	303	7,748	227	-51.08%	4.79	-25.08%	1.14
clang-23353	750	98	1,025	91	1,392	82	-16.33%	1.86	-9.89%	1.36
clang-25900	915	239	1,676	104	3,136	193	-19.25%	3.43	85.58%	1.87
clang-26760	1,718	120	2,384	56	2,263	52	-56.67%	1.32	-7.14%	0.95
clang-27137	7,391	180	9,962	88	13,137	83	-53.89%	1.78	-5.68%	1.32
clang-27747	1,013	117	1,759	79	2,149	83	-29.06%	2.12	5.06%	1.22
clang-31259	1,719	406	11,695	282	13,375	249	-38.67%	7.78	-11.70%	1.14
gcc-59903	3,291	308	4,743	198	6,236	158	-48.70%	1.89	-20.20%	1.31
gcc-60116	2,200	443	6,945	247	9,373	211	-52.37%	4.26	-14.57%	1.35
gcc-61383	2,128	272	10,766	195	17,711	183	-32.72%	8.32	-6.15%	1.65
gcc-61917	1,064	150	1,650	103	1,865	70	-53.33%	1.75	-32.04%	1.13
gcc-64990	2,508	239	4,416	203	5,634	165	-30.96%	2.25	-18.72%	1.28
gcc-65383	827	153	1,531	84	5,523	119	-22.22%	6.68	41.67%	3.61
gcc-66186	2,158	327	16,992	226	10,035	221	-32.42%	4.65	-2.21%	0.59
gcc-66375	2,409	440	11,656	227	16,719	289	-34.32%	6.94	27.31%	1.43
gcc-70127	2,726	301	15,053	230	14,502	188	-37.54%	5.32	-18.26%	0.96
gcc-70586	5,496	157	17,335	94	7,717	75	-52.23%	1.40	-20.21%	0.45
gcc-71626	45	51	123	38	140	35	-31.37%	3.11	-7.89%	1.14
Mean	2,093.05	234.35	6,457.25	150.90	7,070.95	142.85	-36.82%	3.02	-1.86%	1.19
STDEV	1,724.73	130.24	5,746.84	82.89	5,587.87	74.75	0.1329	2.31	0.2673	0.63
rust-111502	71	166	930	157	1,155	160	-3.61%	16.27	1.91%	1.24
rust-112061	2,188	458	17,728	442	14,658	418	-8.73%	6.70	-5.43%	0.83
rust-112213	3,617	736	55,834	635	17,786	549	-25.41%	4.92	-13.54%	0.32
rust-112526	2,900	382	10,932	338	10,407	355	-7.07%	3.59	5.03%	0.95
rust-44800	1,048	467	7,766	284	6,117	307	-34.26%	5.84	8.10%	0.79
rust-66851	4,047	728	31,256	713	39,530	660	-9.34%	9.77	-7.43%	1.26
rust-69039	518	114	5,397	101	6,363	99	-13.16%	12.28	-1.98%	1.18
rust-77002	331	263	1,582	247	1,772	246	-6.46%	5.35	-0.40%	1.12
rust-77320	10	39	153	39	147	9	-76.92%	14.70	-76.92%	0.96
rust-77323	4	13	26	13	86	13	0.00%	21.50	0.00%	3.31
rust-77910	13	34	112	21	451	20	-41.18%	34.69	-4.76%	4.03
rust-77919	28	74	280	62	656	69	-6.76%	23.43	11.29%	2.34
rust-78005	18	102	212	102	282	82	-19.61%	15.67	-19.61%	1.33
rust-78325	5	28	61	26	320	26	-7.14%	64.00	0.00%	5.25
rust-78651	9	17	55	9	99	9	-47.06%	11.00	0.00%	1.80
rust-78652	14	56	182	49	195	56	0.00%	13.93	14.29%	1.07
rust-78655	5	26	107	26	230	26	0.00%	46.00	0.00%	2.15
rust-78720	25	72	351	56	493	56	-22.22%	19.72	0.00%	1.40
rust-91725	284	174	1,700	78	1,315	108	-37.93%	4.63	38.46%	0.77
rust-99830	544	303	3,921	281	3,212	281	-7.26%	5.90	0.00%	0.82
Mean	783.95	212.60	6,929.25	183.95	5,263.70	177.45	-18.71%	12.37	-2.55%	1.33
STDEV	1,302.36	229.65	13,915.68	208.93	9,579.61	193.12	0.1994	15.44	0.2099	1.24

Table 5.2: Evaluation results of C-Reduce, Vulcan, and SFC_{Vulcan} on C and Rust benchmarks. Column % Δ shows the average **percentage change in size** achieved compared to the baseline reducer.

Benchmarks	C-Reduce		Vulcan			SFC _{Vulcan}				
	Time (s)	Size (#)	Time (s)	Size (#)	% Δ in size <i>w.r.t.</i> C-Reduce	Time (s)	Size (#)	% Δ in size <i>w.r.t.</i> Vulcan	Time ratio <i>w.r.t.</i> Vulcan	% Δ in size <i>w.r.t.</i> C-Reduce
clang-22382	1,085	70	1,058	108	54.29%	1,852	100	-7.41%	1.75	42.86%
clang-22704	1,916	42	1,574	62	47.62%	1,731	57	-8.06%	1.10	35.71%
clang-23309	3,696	118	6,802	303	156.78%	10,624	211	-30.36%	1.56	78.81%
clang-23353	1,135	74	1,025	91	22.97%	1,496	90	-1.10%	1.46	21.62%
clang-25900	1,687	90	1,676	104	15.56%	2,528	96	-7.69%	1.51	6.67%
clang-26760	2,811	43	2,384	56	30.23%	2,807	52	-7.14%	1.18	20.93%
clang-27137	9,477	50	9,962	88	76.00%	15,185	83	-5.68%	1.52	66.00%
clang-27747	1,963	68	1,759	79	16.18%	2,471	74	-6.33%	1.40	8.82%
clang-31259	4,772	168	11,695	282	67.86%	19,341	248	-12.06%	1.65	47.62%
gcc-59903	6,326	105	4,743	198	88.57%	6,849	129	-34.85%	1.44	22.86%
gcc-60116	5,174	168	6,945	247	47.02%	10,853	194	-21.46%	1.56	15.48%
gcc-61383	3,934	84	10,766	195	132.14%	18,134	137	-29.74%	1.68	63.10%
gcc-61917	3,316	65	1,650	103	58.46%	2,471	74	-28.16%	1.50	13.85%
gcc-64990	4,961	68	4,416	203	198.53%	6,888	141	-30.54%	1.56	107.35%
gcc-65383	2,206	51	1,531	84	64.71%	5,492	74	-11.90%	3.59	45.10%
gcc-66186	5,045	115	16,992	226	96.52%	23,779	208	-7.96%	1.40	80.87%
gcc-66375	7,414	56	11,656	227	305.36%	19,525	202	-11.01%	1.68	260.71%
gcc-70127	6,316	84	15,053	230	173.81%	21,406	187	-18.70%	1.42	122.62%
gcc-70586	6,342	130	17,335	94	-27.69%	31,316	92	-2.13%	1.81	-29.23%
gcc-71626	292	46	123	38	-17.39%	162	35	-7.89%	1.32	-23.91%
Mean	3,993.40	84.75	6,457.25	150.90	80.38%	10,245.50	124.20	-14.51%	1.56	50.39%
STDEV	2,421.34	38.32	5,746.84	82.89	79.90%	9,205.30	63.14	0.1073	0.49	63.10%
rust-111502	822	161	930	157	-2.48%	2,370	156	-0.64%	2.55	-3.11%
rust-112061	4,892	447	17,728	442	-1.12%	25,980	413	-6.56%	1.47	-7.61%
rust-112213	4,413	726	55,834	635	-12.53%	70,048	536	-15.59%	1.25	-26.17%
rust-112526	5,921	537	10,932	338	-37.06%	18,530	334	-1.18%	1.70	-37.80%
rust-44800	6,436	471	7,766	284	-39.70%	10,731	284	0.00%	1.38	-39.70%
rust-66851	5,673	649	31,256	713	9.86%	68,837	658	-7.71%	2.20	1.39%
rust-69039	649	109	5,397	101	-7.34%	10,379	97	-3.96%	1.92	-11.01%
rust-77002	2,097	264	1,582	247	-6.44%	3,428	246	-0.40%	2.17	-6.82%
rust-77320	147	39	153	39	0.00%	367	9	-76.92%	2.40	-76.92%
rust-77323	25	13	26	13	0.00%	128	13	0.00%	4.92	0.00%
rust-77910	88	23	112	21	-8.70%	373	20	-4.76%	3.33	-13.04%
rust-77919	422	70	280	62	-11.43%	699	59	-4.84%	2.50	-15.71%
rust-78005	308	75	212	102	36.00%	603	82	-19.61%	2.84	9.33%
rust-78325	122	34	61	26	-23.53%	442	26	0.00%	7.25	-23.53%
rust-78651	76	12	55	9	-25.00%	150	9	0.00%	2.73	-25.00%
rust-78652	176	49	182	49	0.00%	384	49	0.00%	2.11	0.00%
rust-78655	93	26	107	26	0.00%	450	26	0.00%	4.21	0.00%
rust-78720	476	51	351	56	9.80%	861	53	-5.36%	2.45	3.92%
rust-91725	2,387	101	1,700	78	-22.77%	2,341	77	-1.28%	1.38	-23.76%
rust-99830	4,278	164	3,921	281	71.34%	6,350	269	-4.27%	1.62	64.02%
Mean	1,975.05	201.05	6,929.25	183.95	-3.55%	11,172.55	170.80	-7.65%	2.35	-11.58%
STDEV	2,341.12	231.04	13,915.68	208.93	24.54%	21,095.72	190.52	0.1715	1.44	26.68%

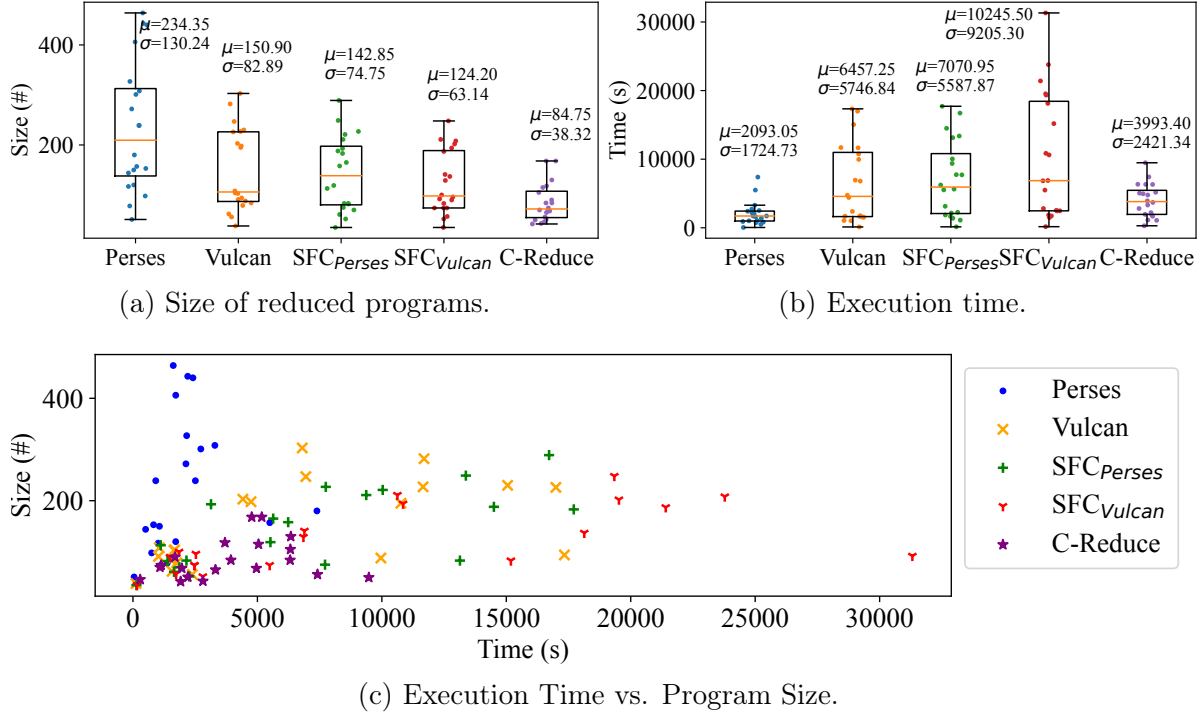


Figure 5.4: Evaluation results of Perses, Vulcan, SFC_{Perses} and SFC_{Vulcan} on C benchmarks.

RQ2: Despite introducing some time overhead, SFC-based reduction methods can significantly enhance the canonicalization capabilities of Perses and Vulcan (with T-Rec integrated). Specifically, SFC_{Perses} and SFC_{Vulcan} eliminate 442 and 435 more duplicates than Perses and Vulcan, respectively.

5.4.3 RQ3: Ablation Studies

To investigate whether each of the proposed SFC-based reduction methods is effective, we conducted ablation studies. Specifically, we implemented six variants of SFC_{Perses} and SFC_{Vulcan} as follows. SFC_{Perses}^{--ie}, SFC_{Perses}^{--ssr}, and SFC_{Perses}^{--sc} are variants of SFC_{Perses} that do not have the auxiliary reducer corresponding to Identifier Elimination, Smaller Structure Replacement, and Structure Canonicalization integrated, respectively. SFC_{Vulcan}^{--ie}, SFC_{Vulcan}^{--ssr}, and SFC_{Vulcan}^{--sc} are variants of SFC_{Vulcan} without these three auxiliary reducer, respectively.

Table 5.3: Evaluation results of Perses, $\text{SFC}_{\text{Perses}}$, $\text{SFC}_{\text{Perses}^{\text{--sc}}}$, Vulcan, $\text{SFC}_{\text{Vulcan}}$, and $\text{SFC}_{\text{Vulcan}^{\text{--sc}}}$ on **Benchmark-Cano**. Each reducer is integrated with T-Rec. The column **Eliminated** shows the total number of eliminated duplicates. The column **Size** and **Time** shows the average size of the outputs and the average execution time.

Reducer	Crash			Miscompilation		
	Eliminated (#)	Size (#)	Time (s)	Eliminated (#)	Size (#)	Time (s)
Perses	1,297/2,501	55.73	146.58	0/1,295	228.14	769.03
$\text{SFC}_{\text{Perses}}$	1,736/2,501	39.59	248.31	3/1,295	168.77	2367.54
$\text{SFC}_{\text{Perses}^{\text{--sc}}}$	1,382/2,501	41.32	217.75	1/1,295	169.99	2001.45
Vulcan	1,327/2,501	40.81	293.68	1/1,295	188.56	2896.75
$\text{SFC}_{\text{Vulcan}}$	1,752/2,501	36.16	312.42	11/1,295	160.88	3914.47
$\text{SFC}_{\text{Vulcan}^{\text{--sc}}}$	1,397/2,501	37.83	284.13	8/1,295	156.28	4245.39

Table 5.4: Results of $\text{SFC}_{\text{Perses}^{\text{--ie}}}$, $\text{SFC}_{\text{Perses}^{\text{--ssr}}}$, $\text{SFC}_{\text{Vulcan}^{\text{--ie}}}$, and $\text{SFC}_{\text{Vulcan}^{\text{--ssr}}}$ on **Benchmark-Reduce**. Column **Worse** is the number of cases where the removal causes a deterioration in minimization capability. Column $\%\Delta$ is the **average percentage increase in size** caused by the removal.

Reducer	C			Rust			SMT-LIBv2		
	Size	Worse	$\%\Delta$ in Size	Size	Worse	$\%\Delta$ in Size	Size	Worse	$\%\Delta$ in Size
$\text{SFC}_{\text{Perses}^{\text{--ie}}}$	192.70	20/20	39.03%	194.05	10/20	7.21%	266.23	169/205	93.91%
$\text{SFC}_{\text{Perses}^{\text{--ssr}}}$	165.90	20/20	16.70%	190.65	17/20	29.90%	145.17	167/205	16.31%
$\text{SFC}_{\text{Vulcan}^{\text{--ie}}}$	132.35	12/20	5.22%	171.35	2/20	0.25%	113.83	29/205	1.11%
$\text{SFC}_{\text{Vulcan}^{\text{--ssr}}}$	141.10	20/20	13.56%	176.05	12/20	18.89%	117.38	137/205	6.03%

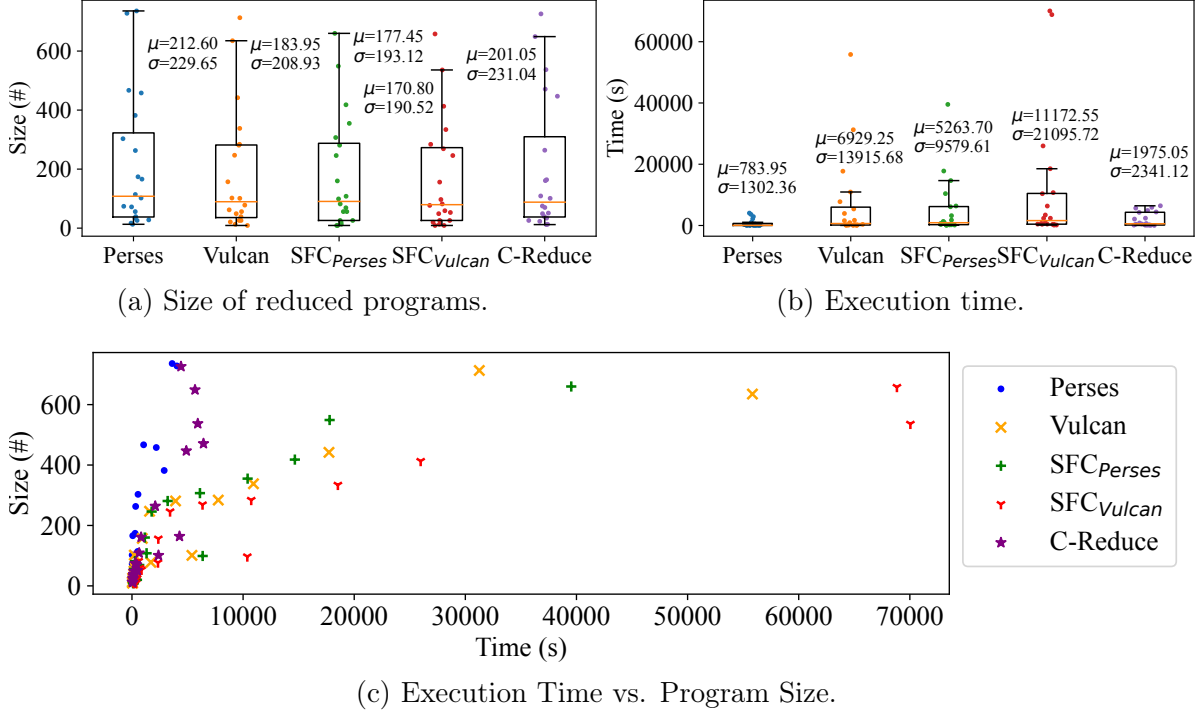


Figure 5.5: Evaluation results of Perses, Vulcan, SFC_{Perses} and SFC_{Vulcan} on Rust benchmarks.

Impacts of Removing Smaller Structure Replacement. Smaller Structure Replacement is proposed to mainly enhance the minimization capabilities of program reducer. Thus, to evaluate its contribution, we compare SFC_{Perses}^{--ssr} and SFC_{Vulcan}^{--ssr} with SFC_{Perses} and SFC_{Vulcan} on **Benchmark-Reduce**, respectively. From Table 5.4, it can be observed that for both SFC_{Perses} and SFC_{Vulcan}, removing Smaller Structure Replacement significantly deteriorates the minimization performance. Specifically, removing Smaller Structure Replacement causes SFC_{Perses} to produce results that are 16.70%, 29.90%, and 16.31% larger on C, Rust, and SMT-LIBv2 benchmarks, respectively. Similarly, removing Smaller Structure Replacement from SFC_{Vulcan} results in outputs that are 13.56%, 18.89%, and 6.03% larger, respectively.

Impacts of Removing Identifier Elimination. Similar to Smaller Structure Replacement, Identifier Elimination is also proposed primarily to boost the minimization performance. Therefore, we evaluate its contribution in the same way we assess Smaller

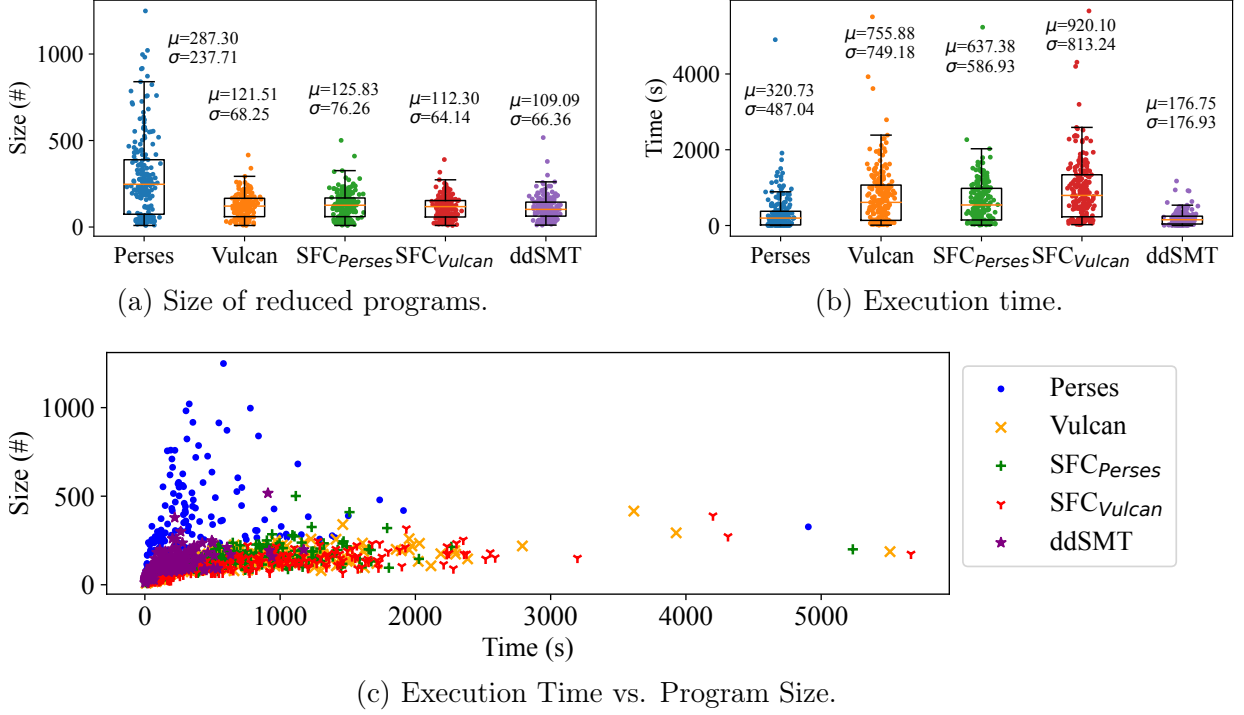


Figure 5.6: Evaluation results of Perses, Vulcan, SFC_{Perses} and SFC_{Vulcan} on SMT-LIBv2 benchmarks.

Structure Replacement. As shown in Table 5.4, removing Identifier Elimination also has a negative impact on the minimization capability of SFC_{Perses}, especially on the SMT-LIBv2 benchmarks. Specifically, the results of SFC_{Perses} without Identifier Elimination are 39.03%, 7.21%, and 93.91% larger on C, Rust, and SMT-LIBv2 benchmarks, respectively. The negative impact of removing Identifier Elimination from SFC_{Vulcan} is not as significant as that of removing it from SFC_{Perses}: the corresponding percentage increases in size of the results are 5.22%, 0.25%, and 1.11%, respectively. The reason behind this may be that Vulcan also has an auxiliary reducer for eliminating identifiers, which is somewhat overlapped with Identifier Elimination. Nevertheless, the increases in size still demonstrate that Identifier Elimination proposed in this chapter is more effective than the auxiliary reducer in Vulcan.

Impacts of Removing Structure Canonicalization. Since Structure Canonicalization is proposed primarily to enhance the canonicalization capability of program reducers, to evaluate the contribution of Structure Canonicalization to the improved canonicalization performance, we compare SFC_{Perses}^{--sc} and SFC_{Vulcan}^{--sc} to SFC_{Perses} and SFC_{Vulcan} on Benchmark-Cano, respectively. As shown in Table 5.3, removing Structure Canonicaliza-

tion from $\text{SFC}_{\text{Perses}}$ leads to 354 and 2 fewer eliminated crash and miscompilation duplicates, respectively. Removing Structure Canonicalization from $\text{SFC}_{\text{Vulcan}}$ also leads to 355 and 3 fewer eliminated crash and miscompilation duplicates, respectively. Such results indicate that Structure Canonicalization significantly contributes to the superior canonicalization performance of both $\text{SFC}_{\text{Perses}}$ and $\text{SFC}_{\text{Vulcan}}$.

RQ3: All the proposed SFC-based reduction methods significantly contribute to the improved reduction performance of $\text{SFC}_{\text{Perses}}$ and $\text{SFC}_{\text{Vulcan}}$. Removing Smaller Structure Replacement or Identifier Elimination can result in large outputs and the ratio is up to 93.91%. Without Structure Canonicalization, more than 300 duplicated crash bugs cannot be eliminated.

5.5 Related Work

This section discusses the studies related to this chapter. *Delta Debugging* is the first systematic work in the direction of bug-triggering test input minimization [226]. This work proposed an algorithm named **ddmin** which is designed to reduce a list of elements. This algorithm still serves as a fundamental reduction algorithm in many advanced reduction techniques proposed by subsequent research studies [152, 112, 192, 221, 161, 114, 229, 235]. ProbDD later improves **ddmin** with a probabilistic model [209]. Zhang *et al.* conducted an in-depth theoretical and empirical analysis of ProbDD, and then proposed Counter-Based Delta Debugging (CDD), a simplified version of ProbDD that reduces the complexity from both theory and implementation perspectives [229]. Zhou *et al.* introduced Weighted Delta Debugging (WDD), a novel concept to leverage the size of elements in partitioning [235].

The algorithm **ddmin** and its variants cannot well handle highly structured inputs on themselves. Targeting to tackle this limitation, *Hierarchical Delta Debugging* (HDD) was proposed by Misherghi and Su [152]. Specifically, HDD performs **ddmin** on each level of the tree representation (*e.g.*, the parse tree) of the input from top to bottom. After HDD was proposed, many researchers improved HDD from different aspects with different mechanisms [95, 97, 112, 208]. Hodovan *et al.* proposed Picieny, which uses extended context-free grammars to modernize the original HDD implementation [95]. In the next year, the same authors proposed another technique called Coarse Hierarchical Delta Debugging (CHDD) to further improve the efficiency of HDD [97]. Kiss *et al.* proposed a recursive version of HDD named HDDr also for saving reduction time [112]. Vince *et al.* improved the performance of HDD by integrating the hoisting operation [208].

Although HDD and its variants can recognize the structure of the input to be reduced, the transformations (*i.e.*, deletions) that they perform do not preserve the syntactic validity, which eventually leads to limited overall reduction performance. To overcome this obstacle, Sun *et al.* proposed Perses, a *syntax-guided program reducer* that always generates programs that conform to the provided formal grammar during the reduction process. Gharachorlu *et al.* further improved the efficiency of Perses by optimizing the order of reducing each portion of the program [77, 79], and Tian *et al.* proposed a better implementation that greatly reduced the engineering work required for applying Perses to different programming languages [198]. Tian *et al.* introduced a novel caching scheme to speed up program reduction [200].

To further trade off efficiency for smaller reduction results, Xu *et al.* proposed Vulcan, a reducer that can switch to more exhaustive reducers whenever the main reducer (*i.e.*, Perses) cannot make any progress [221]. In the study of Vulcan, three auxiliary reducers are proposed, which are Tree-Based Local Exhaustive Enumeration, Identifier Replacement, and Sub-Tree Replacement, respectively. Among these auxiliary reducers, the last two are similar to Identifier Elimination and Smaller Structure Replacement proposed in this chapter. However, compared to Identifier Replacement, Identifier Elimination is more effective as it can eliminate uses of identifiers not only by replacing but also by utilizing SFC. Meanwhile, compared to Smaller Structure Replacement, Sub-Tree Replacement in Vulcan lacks a structure-reusing mechanism, which significantly limits its effectiveness. The extensive evaluation also demonstrates that compared to Vulcan, the methods proposed in this chapter are either more effective or complementary. T-Rec is another syntax-guided program reducer proposed recently [220]. It improves the reduction performance (especially the canonicalization performance) by introducing a fine-grained reduction process, which leverages the lexical syntax to canonicalize tokens. As demonstrated in § 5.4.1, SFC-based reduction methods can significantly enhance the canonicalization capabilities of Perses and Vulcan with T-Rec integrated. Such a result indicates that T-Rec and our approach are complementary to some extent.

There are also many *language-specific program reducers* that are specifically optimized for reducing programs in certain programming languages. These reducers are designed with language-specific knowledge and often require heavy engineering work to implement. C-Reduce is a reducer specifically optimized for reducing C programs [170]. ddSMT and ddSMT2.0 are designed for reducing SMT-LIBv2 programs [161, 114]. J-Reduce focuses on Java bytecode [109, 110], and JS Delta is for reducing JavaScript programs [108]. Zhang *et al.* proposed LPR, the first LLMs-aided technique leveraging LLMs to perform language-specific program reduction for multiple languages [228]. Compared to these language-specific ones, SFC-based reduction is much more general and can be applied

to a wide range of programming languages.

There are also program reducers designed for software debloating. For example, CHISEL is a reducer for debloating C/C++ programs [92]. It leverages a reinforcement learning-based approach to improve the efficiency and scalability. Additionally, Bruce *et al.* proposed JShrink, an end-to-end framework for debloating Java bytecode [37].

Instead of directly reducing the test case, some reducers reduce test cases in a different manner. Hypothesis reducer reduces test cases by minimizing the choice sequence with test case generators [144]. PPR, proposed by Zhang *et al.*, reduces pairs of programs, *i.e.*, a seed and a bug-triggering mutant derived from the seed, in mutation-based fuzzing [230]. It reduces both the difference and the common parts of the two programs to facilitate debugging. While the reducer in spirv-fuzz also reduces the difference between seed and mutant, it focuses on reducing the transformation sequences that mutate the seed to the mutant [65]. These works are all complementary to our work.

5.6 Chapter Conclusion

In this chapter, we boost the performance of language-agnostic program reduction by proposing a novel syntax-guided transformation named Structure Form Conversion (SFC). SFC complements the previously proposed two transformations, Compatible Substructure Hoisting and Quantified Node Reduction. Building on SFC, we further propose three reduction methods: Smaller Structure Replacement, Identifier Elimination, and Structure Canonicalization, to effectively and efficiently leverage SFC for program reduction. Through extensive evaluation with two benchmark suites, we demonstrate the proposed three reduction methods can significantly improve both the minimization and canonicalization performance. Specifically, $\text{SFC}_{\text{Perses}}$ produces 36.82%, 18.71%, and 41.05% smaller results than Perses on average on the C, Rust, and SMT-LIBv2 benchmarks, respectively. Similarly, the average results of $\text{SFC}_{\text{Vulcan}}$ are 14.51%, 7.65%, and 7.66% smaller than Vulcan. Furthermore, in **Benchmark-Cano** SFC-based reduction methods help Perses and Vulcan further eliminate 442 and 435 more duplicates (programs that trigger the same bug), respectively.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This thesis advances the state of the art in language-agnostic program reduction by substantially improving both minimization and canonicalization, thereby reducing the long-standing performance gap between AGRs and SPRs. Although AGRs provide broad applicability across diverse languages, their traditional limitations in supported transformations and token-level granularity have hindered their ability to produce minimal and canonical results. The contributions of this thesis address these challenges through a series of complementary techniques.

First, this work introduces Vulcan, a framework that augments a main AGR with auxiliary reducers capable of performing diverse, strategically selected transformations. By enabling iterative cooperation between the main reducer and auxiliary reducers, Vulcan overcomes the stagnation commonly encountered after obtaining a 1-minimal program. The framework unlocks new opportunities for continued reduction and demonstrates significant improvements over the state-of-the-art AGR, Perses, across C, Rust, and SMT-LIBv2 programs.

Second, to strengthen canonicalization—a key feature for facilitating bug deduplication—this thesis proposes T-Rec, a fine-grained, lexical syntax-guided reduction technique. By operating within tokens rather than treating them as atomic, T-Rec reduces superficial differences among programs that essentially trigger the same bug. When integrated into Vulcan, T-Rec successfully eliminates over a thousand additional duplicates and further shrinks reduced programs, demonstrating its substantial impact on both canonicalization and minimization.

Third, this thesis presents SFC, a new syntax-guided transformation, along with three reduction methods designed to utilize it effectively. These SFC-based techniques enrich the search space, enabling additional structural simplifications and canonicalization. When incorporated into Vulcan alongside T-Rec, these methods contribute further size reductions and eliminate additional duplicates, underscoring their effectiveness in enhancing both minimization and canonicalization.

Collectively, the contributions of Vulcan, T-Rec, and the SFC-based transformations significantly elevate the capabilities of language-agnostic reduction. When applied to the state-of-the-art AGR, Perses, they yield programs that are, on average, 43% smaller in **Benchmark-Reduce** and enable the removal of 1,750 more duplicates in **Benchmark-Cano**.

6.2 Future Work

Several avenues remain open for further advancing program reduction.

First, the rise of Large Language Models (LLMs) presents promising opportunities. Recent work has shown that LLMs can substantially improve the reduction performance of AGRs while preserving their language-generic nature [228]. In that study, the authors create additional reduction opportunities by instructing an LLM to perform transformations that, although common across languages, typically require language-specific knowledge to apply. From my perspective, LLMs could support program reduction in even broader ways. For example, instead of directing an LLM to execute predefined transformations, one might prompt it to design a set of transformations that are especially useful for reduction. Systematically exploring such LLM-driven transformation discovery, as well as other potential integration strategies, represents a promising direction for future research.

Second, a major limitation of existing AGRs—including the approaches proposed in this thesis—is their reliance on the availability of a formal grammar for the target language. Without access to such syntax, most transformations become infeasible, significantly hindering effective reduction. A natural research direction, therefore, is to investigate how to enable strong reduction capabilities when the formal grammar is missing or incomplete. One potential solution is to apply grammar induction techniques to synthesize an approximate grammar from a small corpus of programs. Although grammar induction has been widely studied, its application to program reduction remains underexplored. Investigating how induced grammars can support diverse transformations—or how AGRs can robustly operate under imperfect syntax—could meaningfully broaden the applicability of language-agnostic reduction.

References

- [1] ACE. SuperTest compiler test and validation suite. <http://www.ace.nl/compiler/supertest.html>, accessed: 2016-03-20.
- [2] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, & tools*. Pearson Education India, 2007.
- [3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [4] Bertrand Anckaert, Frederik Vandeputte, Bruno Bus, Bjorn Sutter, and Koen Bosschere. Link-time optimization of ia64 binaries. In Marco Danelutto, Marco Vanneschi, and Domenico Laforenza, editors, *Euro-Par 2004 Parallel Processing*, volume 3149 of *Lecture Notes in Computer Science*, pages 284–291. Springer Berlin Heidelberg, 2004.
- [5] Dimitris Andreou. Objectexplorer, 2015. <https://github.com/DimitrisAndreou/memory-measurer>.
- [6] Christophe Andrieu, Nando de Freitas, Arnaud Doucet, and Michael I. Jordan. An Introduction to MCMC for Machine Learning. *Machine Learning*, 50(1):5–43, January 2003.
- [7] ANTLR. The antlr parser generator, 2017.
- [8] Andrew W. Appel and Sandrine Blazy. Separation logic for small-step Cminor. In *Theorem Proving in Higher Order Logics, 20th Int. Conf. TPHOLs 2007*, volume 4732 of *Lecture Notes in Computer Science*, pages 5–21. Springer, 2007.
- [9] Andrew W. Appel and Sandrine Blazy. Separation logic for small-step Cminor (extended version). Research report 6138, INRIA, 2007. 29 pages.

- [10] Andrew W. Appel and Xavier Leroy. A list-machine benchmark for mechanized metatheory. Research report 5914, INRIA, 2006.
- [11] Andrew W. Appel and Xavier Leroy. A list-machine benchmark for mechanized metatheory (extended abstract). In *Proc. Int. Workshop on Logical Frameworks and Meta-Languages (LFMTP'06)*, volume 174/5 of *Electronic Notes in Computer Science*, pages 95–108, 2007.
- [12] Cyrille Artho. Iterative delta debugging. *International Journal on Software Tools for Technology Transfer*, 13(3):223–246, 2011.
- [13] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. NAUTILUS: fishing for deep bugs with grammars. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.
- [14] Antoine Balestrat. CCG: A random C code generator, 2012. <https://github.com/Merkil/ccg/>, accessed: 2022-10-28.
- [15] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 394–403, 2006.
- [16] Sorav Bansal and Alex Aiken. Binary translation using peephole superoptimizers. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 177–192, 2008.
- [17] Ricardo Bedin França, Sandrine Blazy, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. Formally verified optimizing compilation in ACG-based flight control software. In *Embedded Real Time Software and Systems (ERTS 2012)*, 2012.
- [18] Ricardo Bedin França, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. Towards optimizing certified compilation in flight control software. In *Workshop on Predictability and Performance in Embedded Systems (PPES 2011)*, volume 18 of *OpenAccess Series in Informatics*, pages 59–68. Dagstuhl Publishing, 2011.
- [19] Yves Bertot. Extending the calculus of constructions with Tarski’s fix-point theorem. 2006.

- [20] Yves Bertot. Theorem proving support in programming language semantics. Research report 6242, INRIA, 2007.
- [21] Yves Bertot. Structural abstract interpretation: A formal study using Coq. In Ana Bove and Jorge Sousa Pinto, editors, *Language Engineering and Rigorous Software Development, International LerNet ALFA Summer School 2008*, volume 5520 of *Lecture Notes in Computer Science*, pages 153–194. Springer, 2009.
- [22] Yves Bertot, Benjamin Grégoire, and Xavier Leroy. A structured approach to proving compiler optimizations based on dataflow analysis. In *Types for Proofs and Programs, Workshop TYPES 2004*, volume 3839 of *Lecture Notes in Computer Science*, pages 66–81. Springer, 2006.
- [23] Yves Bertot and Vladimir Komendantsky. Fixed point semantics and partial recursion in Coq. In *10th int. ACM SIGPLAN conference on Principles and Practice of Declarative Programming (PPDP 2008)*, pages 89–96. ACM Press, 2008.
- [24] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. Orbs: Language-independent program slicing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 109–120, 2014.
- [25] David Binkley, Nicolas Gold, Mark Harman, Syed S. Islam, Jens Krinke, and Shin Yoo. ORBS: language-independent program slicing. In *Proceedings of the 2014 ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 109–120, 2014.
- [26] Sandrine Blazy. Comment gagner confiance en C ? *Technique et Science Informatiques*, 26(9):1195–1200, 2007.
- [27] Sandrine Blazy. Experiments in validating formal semantics for C. In *Proceedings of the C/C++ Verification Workshop*, pages 95–102. Technical report ICIS-R07015, Radboud University Nijmegen, 2007.
- [28] Sandrine Blazy. *Sémantiques formelles*. Habilitation à diriger les recherches, Université Évry Val d’Essone, October 2008.
- [29] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *FM 2006: Int. Symp. on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer, 2006.

- [30] Sandrine Blazy and Xavier Leroy. Formal verification of a memory model for C-like imperative languages. In *International Conference on Formal Engineering Methods (ICFEM 2005)*, volume 3785 of *Lecture Notes in Computer Science*, pages 280–299. Springer, 2005.
- [31] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.
- [32] Sandrine Blazy and Benoît Robillard. Live-range unsplitting for faster optimal coalescing. In *Proceedings of the ACM SIGPLAN/SIGBED 2009 conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2009)*, pages 70–79. ACM Press, 2009.
- [33] Sandrine Blazy, Benoît Robillard, and Eric Soutif. Coloration avec préférences dans les graphes triangulés. In *Journées Graphes et Algorithmes (JGA’07)*, November 2007.
- [34] Sandrine Blazy, Benoît Robillard, and Eric Soutif. Vérification formelle d’un algorithme d’allocation de registres par coloration de graphes. In *Journées Francophones des Langages Applicatifs (JFLA’08)*, pages 31–46, Étretat, France, January 2008.
- [35] Sandrine Blazy, Benoît Robillard, and Éric Soutif. Coloration avec préférences: complexité, inégalités valides et vérification formelle. In *ROADEF’08, 9e congrès de la Société Française de Recherche Opérationnelle et d’Aide à la Décision*, 2008.
- [36] Richard Bonichon, David Delahaye, and Damien Doligez. Zenon: an extensible automated theorem prover producing checkable proofs. In *Logic for Programming, Artificial Intelligence and Reasoning, 14th Int. Conf. LPAR 2007*, volume 4790 of *Lecture Notes in Artificial Intelligence*, pages 151–165. Springer, 2007.
- [37] Bobby R Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. 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*, pages 135–146, 2020.
- [38] GCC Bugzilla. Bug 66186, 2015.
- [39] GCC Bugzilla. Bug 66691, 2015.

- [40] LLVM Bugzilla. Bug 21467 - clang hangs on valid code at -os and above on x86_64-linux-gnu, 2014.
- [41] LLVM Bugzilla. Bug llvm-26760, 2016.
- [42] Darius Buntinas, Brice Goglin, David Goodell, Guillaume Mercier, and Stephanie Moreaud. Cache-efficient, intranode, large-message mpi communication with mpich2-nemesis. In *2009 International Conference on Parallel Processing*, pages 462–469. IEEE, 2009.
- [43] Junjie Chen, Jiaqi Han, Peiyi Sun, Lingming Zhang, Dan Hao, and Lu Zhang. 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*, pages 223–234, 2019.
- [44] Junjie Chen, Haoyang Ma, and Lingming Zhang. Enhanced compiler bug isolation via memoized search. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 78–89, 2020.
- [45] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 197–208, 2013.
- [46] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. Coverage-directed differential testing of jvm implementations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '16*, pages 85–99, New York, NY, USA, 2016. ACM.
- [47] Ophelia C Chesley, Xiaoxia Ren, and Barbara G Ryder. Crisp: A debugging tool for java programs. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 401–410. IEEE, 2005.
- [48] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, 1959.
- [49] Nathan Chong, Alastair Donaldson, Andrei Lascu, and Christopher Lidbury. Many-core compiler fuzzing. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015.

- [50] Hong-Tai Chou and David J DeWitt. An evaluation of buffer management strategies for relational database systems. *Algorithmica*, 1(1-4):311–336, 1986.
- [51] James Cogan. The patience diff algorithm, 2017.
- [52] CPython. Bug report, 2022.
- [53] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. Testing static analyzers with randomly generated programs. In *NASA Formal Methods - 4th International Symposium (NFM)*, pages 120–125, 2012.
- [54] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. Testing static analyzers with randomly generated programs. In AlwynE. Goodloe and Suzette Person, editors, *NASA Formal Methods*, volume 7226 of *Lecture Notes in Computer Science*, pages 120–125. Springer Berlin Heidelberg, 2012.
- [55] Shaul Dar, Michael J Franklin, Bjorn T Jonsson, Divesh Srivastava, Michael Tan, et al. Semantic data caching and replacement. In *VLDB*, volume 96, pages 330–341, 1996.
- [56] Zaynah Dargaye. Décurryfication certifiée. In *Journées Francophones des Langages Applicatifs (JFLA’07)*, pages 119–134. INRIA, 2007.
- [57] Zaynah Dargaye. *Vérification formelle d’un compilateur pour langages fonctionnels*. PhD thesis, Université Paris 7 Diderot, July 2009.
- [58] Zaynah Dargaye and Xavier Leroy. Mechanized verification of CPS transformations. In *Logic for Programming, Artificial Intelligence and Reasoning, 14th Int. Conf. LPAR 2007*, volume 4790 of *Lecture Notes in Artificial Intelligence*, pages 211–225. Springer, 2007.
- [59] Zaynah Dargaye and Xavier Leroy. A verified framework for higher-order uncurrying optimizations. *Higher-Order and Symbolic Computation*, 22(3):199–231, 2009.
- [60] Bjorn De Sutter, Ludo Van Put, Dominique Chagnet, Bruno De Bus, and Koen De Bosschere. Link-time compaction and optimization of arm executables. *ACM Trans. Embed. Comput. Syst.*, 6(1), February 2007.
- [61] Peter Deutsch and Jean-Loup Gailly. Zlib compressed data format specification version 3.3. Technical report, 1996.

- [62] Alastair Donaldson and David MacIver. Test Case Reduction: Beyond Bugs, May 2021.
- [63] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. Automated testing of graphics shader compilers. *Proc. ACM Program. Lang.*, 1(OOPSLA):93:1–93:29, 2017.
- [64] Alastair F. Donaldson and Andrei Lascu. Metamorphic testing for (graphics) compilers. In *1st International Workshop on Metamorphic Testing, in conjunction with the 38th International Conference on Software Engineering*, 5 2016.
- [65] Alastair F Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpiński. 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*, pages 1017–1032, 2021.
- [66] Chucky Ellison and Grigore Rosu. An executable formal semantics of C with applications. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 533–544, 2012.
- [67] Karine Even-Mendoza, Arindam Sharma, Alastair F Donaldson, and Cristian Cadar. Grayc: Greybox fuzzing of compilers and analysers for c. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1219–1231, 2023.
- [68] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 313–324, 2014.
- [69] Beat Fluri, Michael Wursch, Martin Pinzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on software engineering*, 33(11):725–743, 2007.
- [70] GCC. A guide to testcase reduction, 2020.
- [71] GCC. How to minimize test cases for bugs, 2024.
- [72] GCC Wiki. Finding miscompilations on large testcases. http://gcc.gnu.org/wiki/Analysing_Large_Testcases/.

- [73] GCC Wiki. Predictive Commoning. <https://gcc.gnu.org/wiki/PredictiveCommoning>, accessed: 2016-03-20.
- [74] GCC-Wiki. A guide to testcase reduction, 2020.
- [75] Golnaz Gharachorlu and Nick Sumner. Avoiding the familiar to speed up test case reduction. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 426–437. IEEE, 2018.
- [76] Golnaz Gharachorlu and Nick Sumner. Pardes: Priority aware test case reduction. In *Fundamental Approaches to Software Engineering*, pages 409–426, Cham, 2019. Springer International Publishing.
- [77] Golnaz Gharachorlu and Nick Sumner. Pardis: Priority aware test case reduction. In *International Conference on Fundamental Approaches to Software Engineering*, pages 409–426. Springer, 2019.
- [78] Golnaz Gharachorlu and Nick Sumner. Type batched program reduction. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 398–410, 2023.
- [79] Golnaz Gharachorlu and Nick Sumner. Type batched program reduction. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 398–410, New York, NY, USA, 2023. Association for Computing Machinery.
- [80] W. R. Gilks. *Markov Chain Monte Carlo In Practice*. Chapman and Hall/CRC, 1999.
- [81] Stéphane Glondou. Extraction certifiée dans Coq-en-Coq. In *Journées Francophones des Langages Applicatifs (JFLA’09)*, pages 105–118. INRIA, 2009.
- [82] GNU Compiler Collection. Gcov - Using the GNU Compiler Collection (GCC). <http://gcc.gnu.org/onlinedocs/gcc/Gcov.html>, accessed: 2016-03-20.
- [83] Rahul Gopinath, Alexander Kampmann, Nikolas Havrikov, Ezekiel O Soremekun, and Andreas Zeller. Abstracting failure-inducing inputs. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pages 237–248, 2020.
- [84] Rahul Gopinath, Alexander Kampmann, Nikolas Havrikov, Ezekiel O Soremekun, and Andreas Zeller. The ddset github repo, 2020.

- [85] Mark Adler Greg Roelofs. A massively spiffy yet delicately unobtrusive compression library, Jan 1996.
- [86] Sheila A. Greibach. A new normal-form theorem for context-free phrase structure grammars. *J. ACM*, 12(1):42–52, January 1965.
- [87] Alex Groce, Josie Holmes, and Kevin Kellar. One test to rule them all. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, page 1–11, New York, NY, USA, 2017. Association for Computing Machinery.
- [88] Alex Groce, Josie Holmes, and Kevin Kellar. One test to rule them all. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, page 1–11, New York, NY, USA, 2017. Association for Computing Machinery.
- [89] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. Swarm testing. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 78–88, 2012.
- [90] Shay Gueron, Simon Johnson, and Jesse Walker. Sha-512/256. In *2011 Eighth International Conference on Information Technology: New Generations*, pages 354–358. IEEE, 2011.
- [91] Jim Handy. *The Cache Memory Book (2nd Ed.): The Authoritative Reference on Cache Design*. Academic Press, Inc., USA, 1998.
- [92] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective program debloating via reinforcement learning. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 380–394. ACM, 2018.
- [93] Satia Herfert, Jibesh Patra, and Michael Pradel. Automatically reducing tree-structured test inputs. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017*, page 861–871. IEEE Press, 2017.
- [94] Tony Hoare. The verifying compiler: A grand challenge for computing research. In *Modular Programming Languages*, pages 25–35. Springer, 2003.

- [95] Renáta Hodován and Ákos Kiss. Modernizing hierarchical delta debugging. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*, A-TEST 2016, page 31–37, New York, NY, USA, 2016. Association for Computing Machinery.
- [96] Renáta Hodován and Ákos Kiss. Practical improvements to the minimizing delta debugging algorithm. In *International Conference on Software Engineering and Applications*, volume 2, pages 241–248. SciTePress, 2016.
- [97] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. Coarse hierarchical delta debugging. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*, pages 194–203. IEEE Computer Society, 2017.
- [98] Renata Hodovan, Akos Kiss, and Tibor Gyimothy. Tree Preprocessing and Test Outcome Caching for Efficient Hierarchical Delta Debugging. *2017 IEEE/ACM 12th International Workshop on Automation of Software Testing (AST)*, 2017.
- [99] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. Grammarinator: a grammar-based open source fuzzer. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, A-TEST 2018, page 45–48, New York, NY, USA, 2018. Association for Computing Machinery.
- [100] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458, 2012.
- [101] IBM. The T.J. Watson Libraries for Analysis, 2017.
- [102] JavaCC. The Java Parser Generator, 2017. <https://javacc.org/>, accessed: 2017-08-05.
- [103] JerryScript. Bug report, 2022.
- [104] Jesse Ruderman. Introducing jsfunfuzz.
- [105] Theodore Johnson, Dennis Shasha, et al. 2q: a low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 439–450. Citeseer, 1994.
- [106] Rajeev Joshi, Greg Nelson, and Keith H. Randall. Denali: A goal-directed super-optimizer. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 304–314, 2002.

- [107] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. Validating LR(1) parsers. In *Programming Languages and Systems – 21st European Symposium on Programming, ESOP 2012*, volume 7211 of *Lecture Notes in Computer Science*, pages 397–416. Springer, 2012.
- [108] JS Delta. JS Delta, 2017.
- [109] Christian Gram Kalhauge and Jens Palsberg. Binary reduction of dependency graphs. In Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo, editors, *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, pages 556–566. ACM, 2019.
- [110] Christian Gram Kalhauge and Jens Palsberg. Logical bytecode reduction. In Stephen N. Freund and Eran Yahav, editors, *PLDI ’21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 1003–1016. ACM, 2021.
- [111] J. Kelsey, Shu jen H. Chang, and Ray A. Perlner. Sha-3 derived functions: cshake, kmac, tuplehash, and parallelhash. 2016.
- [112] Ákos Kiss, Renáta Hodován, and Tibor Gyimóthy. 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, A-TEST 2018*, page 16–22, New York, NY, USA, 2018. Association for Computing Machinery.
- [113] Ákos Kiss, Renáta Hodován, and Tibor Gyimóthy. 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, A-TEST 2018*, page 16–22, New York, NY, USA, 2018. Association for Computing Machinery.
- [114] Gereon Kremer, Aina Niemetz, and Mathias Preiner. ddsmt 2.0: Better delta debugging for the smt-libv2 language and friends. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*, volume 12760 of *Lecture Notes in Computer Science*, pages 231–242. Springer, 2021.
- [115] Patrick Kreutzer, Stefan Kraus, and Michael Philippsen. Language-agnostic generation of compilable test programs. In *13th IEEE International Conference on*

Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020, pages 39–50. IEEE, 2020.

- [116] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. Proving optimizations correct using parameterized program equivalence. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 327–337, 2009.
- [117] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. pages 216–226, 2014.
- [118] Vu Le, Chengnian Sun, and Zhendong Su. Randomized stress-testing of link-time optimizers. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*, 2014.
- [119] Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. *ACM SIGPLAN Notices*, 50(10):386–399, 2015.
- [120] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: large-scale detection of dom-based XSS. In *CCS*, pages 1193–1204, 2013.
- [121] Xavier Leroy. Coinductive big-step operational semantics. In *European Symposium on Programming (ESOP’06)*, volume 3924 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 2006.
- [122] Xavier Leroy. 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*, pages 42–54, 2006.
- [123] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 42–54, 2006.
- [124] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
- [125] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
- [126] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

- [127] Xavier Leroy. Comment faire confiance à un compilateur? *La Recherche*, 440, April 2010. Les cahiers de l’INRIA.
- [128] Xavier Leroy. Mechanized semantics. In *Logics and languages for reliability and security*, volume 25 of *NATO Science for Peace and Security Series D: Information and Communication Security*, pages 195–224. IOS Press, 2010.
- [129] Xavier Leroy. Verified squared: does critical software deserve verified tools? In *38th symposium Principles of Programming Languages*, pages 1–2. ACM Press, 2011. Abstract of invited lecture.
- [130] Xavier Leroy. Mechanized semantics for compiler verification. In Ranjit Jhala and Atsushi Igarashi, editors, *Programming Languages and Systems, 10th Asian Symposium, APLAS 2012*, volume 7705 of *Lecture Notes in Computer Science*, pages 386–388. Springer, 2012. Abstract of invited talk.
- [131] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. The CompCert memory model, version 2. Research report RR-7987, INRIA, June 2012.
- [132] Xavier Leroy and Sandrine Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31, 2008.
- [133] Pierre Letouzey. Extraction in Coq: An overview. In *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008*, volume 5028 of *Lecture Notes in Computer Science*, pages 359–369. Springer, 2008.
- [134] Alan Leung, Manish Gupta, Yuvraj Agarwal, Rajesh Gupta, Ranjit Jhala, and Sorin Lerner. Verifying GPU kernels by test amplification. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 383–394, 2012.
- [135] Cong Li, Yanyan Jiang, Chang Xu, and Zhendong Su. Validating jit compilers via compilation space exploration. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP ’23*, page 66–79, New York, NY, USA, 2023. Association for Computing Machinery.
- [136] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F Donaldson. Many-core compiler fuzzing. *ACM SIGPLAN Notices*, 50(6):65–76, 2015.

- [137] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Commun. ACM*, 21(6):466–471, June 1978.
- [138] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. Incbricks: Toward in-network computation with an in-network cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 795–809, 2017.
- [139] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random testing for c and c++ compilers with yarpngen. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–25, 2020.
- [140] LLVM. How to submit an llvm bug report, 2022.
- [141] LLVM/Clang. Clang documentation – libtooling, 2022.
- [142] LLVM’s Analysis and Transform Passes. Loop-Closed SSA Form Pass. <http://llvm.org/docs/Passes.html#lcssa-loop-closed-ssa-form-pass>, accessed: 2016-03-20.
- [143] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 22–32, New York, NY, USA, 2015. ACM.
- [144] David R MacIver and Alastair F Donaldson. Test-case reduction via test-case generation: Insights from the hypothesis reducer (tool insights paper). In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2020.
- [145] Mohammad Ali Maddah-Ali and Urs Niesen. Fundamental limits of caching. *IEEE Transactions on information theory*, 60(5):2856–2867, 2014.
- [146] David Majnemer. Instcombine: Remove infinite loop caused by foldopintophi, 2014.
- [147] Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Toward a verified relational database management system. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 237–248, 2010.

- [148] Lorenzo Martignoni, Roberto Paleari, Alessandro Reina, Giampaolo Fresi Roglia, and Danilo Bruschi. A methodology for testing CPU emulators. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(4):29, 2013.
- [149] Henry Massalin. Superoptimizer – A look at the smallest program. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 122–126, 1987.
- [150] William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.
- [151] Scott McPeak, Daniel S. Wilkerson, and Simon Goldsmith. Berkeley Delta, 2003.
- [152] Ghassan Misherghi and Zhendong Su. HDD: hierarchical delta debugging. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pages 142–151. ACM, 2006.
- [153] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. Compiler testing via a theory of sound optimisations in the c11/c++ 11 memory model. *ACM SIGPLAN Notices*, 48(6):187–196, 2013.
- [154] MozillaSecurity. Using lithium, 2022.
- [155] Eugene W Myers. Ano (nd) difference algorithm and its variations. *Algorithmica*, 1(1):251–266, 1986.
- [156] Eriko Nagai, Hironobu Awazu, Nagisa Ishiura, and Naoya Takeda. Random testing of c compilers targeting arithmetic optimization. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2012)*, pages 48–53, 2012.
- [157] Eriko Nagai, Hironobu Awazu, Nagisa Ishiura, and Naoya Takeda. Random testing of C compilers targeting arithmetic optimization. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2012)*, pages 48–53, 2012.
- [158] Eriko Nagai, Atsushi Hashimoto, and Nagisa Ishiura. Scaling up size and number of expressions in random testing of arithmetic optimization of C compilers. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2013)*, pages 88–93, 2013.

- [159] Eriko Nagai, Atsushi Hashimoto, and Nagisa Ishiura. Reinforcing random testing of arithmetic optimization of c compilers by scaling up size and number of expressions. *IPSSJ Transactions on System LSI Design Methodology*, 7:91–100, 2014.
- [160] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 83–94, 2000.
- [161] Aina Niemetz and Armin Biere. ddsmt: a delta debugger for the smt-lib v2 format. In *Proceedings of the 11th International Workshop on Satisfiability Modulo Theories, SMT*, pages 8–9, 2013.
- [162] Elizabeth J O’neil, Patrick E O’neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306, 1993.
- [163] Mateusz Pawlik and Nikolaus Augsten. Rted: a robust algorithm for the tree edit distance. *arXiv preprint arXiv:1201.0230*, 2011.
- [164] Plum Hall, Inc. The Plum Hall Validation Suite for C. <http://www.plumhall.com/stec.html>, accessed: 2016-03-20.
- [165] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 151–166, 1998.
- [166] John Regehr. Embedded in academia. <http://blog.regehr.org/>.
- [167] John Regehr. Reducers are Fuzzers – EMBEDDED IN ACADEMIA, 2015. <https://blog.regehr.org/archives/1284>.
- [168] John Regehr. [creduce-dev] cache, 2016. <http://www.flux.utah.edu/listarchives/creduce-dev/msg00284.html>, accessed: 2022-10-28.
- [169] John Regehr. Design and evolution of c-reduce, 2019.
- [170] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for c compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’12*, page 335–346, New York, NY, USA, 2012. Association for Computing Machinery.

- [171] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 335–346, 2012.
- [172] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. The c-reducer github repo, 2022.
- [173] Laurence Rideau, Bernard Paul Serpette, and Xavier Leroy. Tilting at windmills with Coq: Formal verification of a compilation algorithm for parallel moves. *Journal of Automated Reasoning*, 40(4):307–326, 2008.
- [174] Silvain Rideau and Xavier Leroy. Validating register allocation and spilling. In *Compiler Construction (CC 2010)*, volume 6011 of *Lecture Notes in Computer Science*, pages 224–243. Springer, 2010.
- [175] Manuel Rigger and Zhendong Su. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2020*, page 1140–1152, New York, NY, USA, 2020. Association for Computing Machinery.
- [176] Valentin Robert and Xavier Leroy. A formally-verified alias analysis. In *Certified Programs and Proofs (CPP 2012)*, volume 7679 of *Lecture Notes in Computer Science*, pages 11–26. Springer, 2012.
- [177] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H Bailey, Costin Iancu, and David Hough. Precimonious: Tuning assistant for floating-point precision. In *SC’13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2013.
- [178] Rust. Rust issues, 2022.
- [179] Rust. Finding a minimal, standalone example, 2024.
- [180] Hanan Samet. *Automatically proving the correctness of translations involving optimized code*. PhD Thesis, Stanford University, May 1975.
- [181] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. FLAX: systematic discovery of client-side validation vulnerabilities in rich web applications. In *NDSS*, 2010.

- [182] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 305–316, 2013.
- [183] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic optimization of floating-point programs with tunable precision. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 53–64, 2014.
- [184] Mozilla Security. Lithium: Line-based testcase reducer, 2008. <https://github.com/MozillaSecurity/lithium>, accessed: 2022-10-28.
- [185] Prashast Srivastava and Mathias Payer. Gramatron: Effective grammar-aware fuzzing. In *Proceedings of the 30th acm sigsoft international symposium on software testing and analysis*, pages 244–256, 2021.
- [186] Standard Performance Evaluation Corporation. SPEC CINT2006 Benchmarks.
- [187] Daniil Stepanov, Marat Akhin, and Mikhail Belyaev. ReduKtor: How we stopped worrying about bugs in kotlin compiler. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 317–326. IEEE, 2019.
- [188] Chengnian Sun, Vu Le, and Zhendong Su. Finding and analyzing compiler warning defects. In *Proceedings of the 38th International Conference on Software Engineering, ICSE ’16*, pages 203–213, 2016.
- [189] Chengnian Sun, Vu Le, and Zhendong Su. Finding compiler bugs via live code mutation. In Eelco Visser and Yannis Smaragdakis, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 849–863. ACM, 2016.
- [190] Chengnian Sun, Vu Le, and Zhendong Su. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016*, pages 849–863, 2016.
- [191] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. Toward understanding compiler bugs in GCC and LLVM. In Andreas Zeller and Abhik Roychoudhury, editors, *Proceedings of the 25th International Symposium on Software Testing and*

Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016, pages 294–305. ACM, 2016.

- [192] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. Perses: Syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering, ICSE '18*, page 361–371, New York, NY, USA, 2018. Association for Computing Machinery.
- [193] Yutian Tang, Hao Zhou, Xiapu Luo, Ting Chen, Haoyu Wang, Zhou Xu, and Yan Cai. Xdeblobat: Towards automated feature-oriented app debloating. *IEEE Transactions on Software Engineering*, 2021.
- [194] Ross Tate, Michael Stepp, and Sorin Lerner. Generating compiler optimizations from proofs. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 389–402, 2010.
- [195] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *Proceedings of the ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, pages 264–276, 2009.
- [196] Zachary Tatlock and Sorin Lerner. Bringing extensibility to verified compilers. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 111–121, 2010.
- [197] The Clang Team. Clang 3.4 documentation: LibTooling. <http://clang.llvm.org/docs/LibTooling.html>, accessed: 2016-03-20.
- [198] Jia Le Tian, Mengxiao Zhang, Zhenyang Xu, Yongqiang Tian, Yiwen Dong, and Chengnian Sun. 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*, page 2137–2141, New York, NY, USA, 2023. Association for Computing Machinery.
- [199] Yongqiang Tian, Zhenyang Xu, Yiwen Dong, Chengnian Sun, and Shing-Chi Cheung. 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*, pages 4873–4882. ijcai.org, 2023.
- [200] Yongqiang Tian, Xueyan Zhang, Yiwen Dong, Zhenyang Xu, Mengxiao Zhang, Yu Jiang, Shing-Chi Cheung, and Chengnian Sun. On the caching schemes to speed up program reduction. *ACM Trans. Softw. Eng. Methodol.*, 33(1), nov 2023.

- [201] Yongqiang Tian, Xueyan Zhang, Yiwen Dong, Zhenyang Xu, Mengxiao Zhang, Yu Jiang, Shing-Chi Cheung, and Chengnian Sun. On the caching schemes to speed up program reduction. *ACM Trans. Softw. Eng. Methodol.*, 33(1):17:1–17:30, 2024.
- [202] Jean-Baptiste Tristan. *Formal verification of translation validators*. PhD thesis, Université Paris 7 Diderot, November 2009.
- [203] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. Evaluating value-graph translation validation for LLVM. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 295–305, 2011.
- [204] Jean-Baptiste Tristan and Xavier Leroy. Formal verification of translation validators: A case study on instruction scheduling optimizations. In *Proceedings of the 35th ACM Symposium on Principles of Programming Languages (POPL)*, pages 17–27. ACM Press, January 2008.
- [205] Jean-Baptiste Tristan and Xavier Leroy. Verified validation of Lazy Code Motion. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 316–326, 2009.
- [206] Jean-Baptiste Tristan and Xavier Leroy. A simple, verified validator for software pipelining. In *Proceedings of the 37th ACM Symposium on Principles of Programming Languages (POPL)*, pages 83–92. ACM Press, 2010.
- [207] Dániel Vince. Iterating the minimizing delta debugging algorithm. In Ákos Kiss, Beatriz Marín, and Mehrdad Saadatmand, editors, *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation, A-TEST 2022, Singapore, Singapore, November 17-18, 2022*, pages 57–60. ACM, 2022.
- [208] Dániel Vince, Renáta Hodován, Daniella Bársony, and Ákos Kiss. Extending hierarchical delta debugging with hoisting. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*, pages 60–69. IEEE, 2021.
- [209] Guancheng Wang, Ruobing Shen, Junjie Chen, Yingfei Xiong, and Lu Zhang. Probabilistic delta debugging. In Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta, editors, *ESEC/FSE ’21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, pages 881–892. ACM, 2021.

- [210] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: grammar-aware grey-box fuzzing. In Joanne M. Atlee, Tefvik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 724–735. IEEE / ACM, 2019.
- [211] Theodore Luo Wang, Yongqiang Tian, Yiwen Dong, Zhenyang Xu, and Chengnian Sun. Compilation consistency modulo debug information. In Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift, editors, *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 146–158. ACM, 2023.
- [212] Wikipedia. Global Value Numbering. https://en.wikipedia.org/wiki/Global_value_numbering, accessed: 2016-03-20.
- [213] Wikipedia. GNU Compiler Collection. https://en.wikipedia.org/wiki/GNU_Compiler_Collection, accessed: 2024-04-11.
- [214] Wikipedia. Loop unswitching. https://en.wikipedia.org/wiki/Loop_unswitching, accessed: 2016-03-20.
- [215] Wikipedia. Sequence point. https://en.wikipedia.org/wiki/Sequence_point, accessed: 2016-03-20.
- [216] Wikipedia. Jaccard index, 2022.
- [217] Frank Wilcoxon. *Individual comparisons by ranking methods*. Springer, 1992.
- [218] Dominik Winterer, Chengyu Zhang, and Zhendong Su. Validating smt solvers via semantic fusion. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020*, page 718–730, New York, NY, USA, 2020. Association for Computing Machinery.
- [219] Yuanmin Xie, Zhenyang Xu, Yongqiang Tian, Min Zhou, Xintong Zhou, and Chengnian Sun. Kitten: A simple yet effective baseline for evaluating llm-based compiler testing techniques. In Mike Papadakis, Myra B. Cohen, and Paolo Tonella, editors, *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA Companion 2025, Clarion Hotel Trondheim, Trondheim, Norway, June 25-28, 2025*, pages 21–25. ACM, 2025.

- [220] Zhenyang Xu, Yongqiang Tian, Mengxiao Zhang, Jiarui Zhang, Puzhuo Liu, Yu Jiang, and Chengnian Sun. T-rec: Fine-grained language-agnostic program reduction guided by lexical syntax. *ACM Trans. Softw. Eng. Methodol.*, aug 2024. Just Accepted.
- [221] Zhenyang Xu, Yongqiang Tian, Mengxiao Zhang, Gaosen Zhao, Yu Jiang, and Chengnian Sun. Pushing the limit of 1-minimality of language-agnostic program reduction. *Proc. ACM Program. Lang.*, 7(OOPSLA1), apr 2023.
- [222] Chen Yang, Junjie Chen, Xingyu Fan, Jiajun Jiang, and Jun Sun. Silent compiler bug de-duplication via three-dimensional analysis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 677–689, New York, NY, USA, 2023. Association for Computing Machinery.
- [223] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’11, page 283–294, New York, NY, USA, 2011. Association for Computing Machinery.
- [224] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 283–294, 2011.
- [225] Shin Yoo, David Binkley, and Roger D. Eastman. Seeing is slicing: Observation based slicing of picture description languages. In *Proceedings of the 2014 IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 175–184, 2014.
- [226] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.
- [227] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. Localizing failure-inducing program edits based on spectrum information. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 23–32. IEEE, 2011.
- [228] Mengxiao Zhang, Yongqiang Tian, Zhenyang Xu, Yiwen Dong, Shin Hwei Tan, and Chengnian Sun. Lpr: Large language models-aided program reduction. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (Vienna, Austria)(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA. <https://doi.org/10.1145/3650212.3652126>, 2024.

- [229] Mengxiao Zhang, Zhenyang Xu, Yongqiang Tian, Xinru Cheng, and Chengnian Sun. Toward a better understanding of probabilistic delta debugging. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*, pages 2024–2035. IEEE, 2025.
- [230] Mengxiao Zhang, Zhenyang Xu, Yongqiang Tian, Yu Jiang, and Chengnian Sun. Ppr: Pairwise program reduction. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 338–349, 2023.
- [231] Qirun Zhang, Chengnian Sun, and Zhendong Su. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 347–361. ACM, 2017.
- [232] Sai Zhang, Yu Lin, Zhongxian Gu, and Jianjun Zhao. 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*, pages 77–83, 2008.
- [233] Chen Zhao, Yunzhi Xue, Qiuming Tao, Liang Guo, and Zhaohui Wang. Automated test program generation for an industrial optimizing compiler. In *ICSE Workshop on Automation of Software Test (AST)*, pages 36–43, 2009.
- [234] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formal verification of SSA-based optimizations for LLVM. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 175–186, 2013.
- [235] Xintong Zhou, Zhenyang Xu, Mengxiao Zhang, Yongqiang Tian, and Chengnian Sun. WDD: weighted delta debugging. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*, pages 1592–1603. IEEE, 2025.