LPR: Large Language Models-Aided Program Reduction

Mengxiao Zhang
m492zhan@uwaterloo.ca
University of Waterloo
Canada

Yongqiang Tian*
yqtian@ust.hk
The Hong Kong University of Science and Technology
China

Zhenyang Xu
zhenyang.xu@uwaterloo.ca
University of Waterloo
Canada

Yiwen Dong
yiwen.dong@uwaterloo.ca
University of Waterloo
Canada

Shin Hwei Tan
shinhwei.tan@concordia.ca
Concordia University
Canada

Chengnian Sun
cnsun@uwaterloo.ca
University of Waterloo
Canada

ABSTRACT

Program reduction is a widely used technique to facilitate debugging compilers by automatically minimizing programs that trigger compiler bugs. Existing program reduction techniques are either generic to a wide range of languages (such as Perses and Vulcan) or specifically optimized for one certain language by exploiting language-specific knowledge (*e.g.*, C-Reduce). However, synergistically combining both generality across languages and optimality to a specific language in program reduction is yet to be explored.

This paper proposes LPR, the first LLMs-aided technique leveraging LLMs to perform language-specific program reduction for multiple languages. The key insight is to utilize both the language generality of program reducers such as Perses and the language-specific semantics learned by LLMs. Concretely, language-generic program reducers can efficiently reduce programs into a small size that is suitable for LLMs to process; LLMs can effectively transform programs via the learned semantics to create new reduction opportunities for the language-generic program reducers to further reduce the programs.

Our thorough evaluation on 50 benchmarks across three programming languages (*i.e.*, C, Rust and JavaScript) has demonstrated LPR's practicality and superiority over Vulcan, the state-of-the-art language-generic program reducer. For effectiveness, LPR surpasses Vulcan by producing 24.93%, 4.47%, and 11.71% smaller programs on benchmarks in C, Rust and JavaScript, separately. Moreover, LPR and Vulcan have the potential to complement each other. For the C language for which C-Reduce is optimized, by applying Vulcan to the output produced by LPR, we can attain program sizes that are on par with those achieved by C-Reduce. For efficiency perceived by users, LPR is more efficient when reducing large and complex programs, taking 10.77%, 34.88%, 36.96% less time than Vulcan to finish all the benchmarks in C, Rust and JavaScript, separately.

*Corresponding author

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3652126

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

Program Reduction, Large Language Models, Program Semantics

ACM Reference Format:

Mengxiao Zhang, Yongqiang Tian, Zhenyang Xu, Yiwen Dong, Shin Hwei Tan, and Chengnian Sun. 2024. LPR: Large Language Models-Aided Program Reduction. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3652126

1 INTRODUCTION

Program reduction techniques [8, 15, 16, 25–27, 33, 38, 39, 48] aim to facilitate compiler debugging by minimizing the bug-triggering programs with efficacy and efficiency. Given a program P and a property ψ that P preserves, program reduction techniques (*a.k.a.*, program reducers) produce a minimal program P_{min} that still preserves ψ. Program reduction has been widely used in various software engineering tasks [7, 12], especially in compiler testing and debugging [18, 23, 36, 37, 42].

However, a critical challenge in program reduction has not been properly addressed, *i.e.*, the trade-off between generality across languages and specificity to a certain language. Currently, there are two categories of program reduction techniques: language-specific reduction [15, 16, 27] and language-generic reduction [25, 33, 48, 50, 52]. The former category leverages language-specific semantics to transform and shrink programs in certain languages, while the latter only uses transformations applicable to any programming language. Although language-specific reducers are usually more effective in reduction, designing an effective reducer for a specific language, especially language-specific transformations, requires a deep understanding of language features and a significant amount of time and engineering effort. Therefore, only a limited set of languages have custom-designed reducers, such as C [27], Java [15, 16], and SMT-LIBv2 [26]. Meanwhile, language-generic reducers can be applied to diverse languages, but lack the knowledge of language features and semantics and thus are incapable of performing language-specific transformations (*e.g.*, function inlining) that can enable further reduction. As a result, they cannot utilize peculiar features of each language to produce optimally reduced programs.

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

Meanwhile, we also notice that recent Large Language Models (LLMs) could be powerful assistants in performing language-specific transformations, like how they perform in other software engineering tasks such as code generation and test generation [6, 10, 11, 14, 34, 43, 46]. Specifically, LLMs are trained with massive programs, and they have started to exhibit the ability to analyze and transform programs in prevalent languages. If we can properly leverage this ability for program reduction, we may have a language-generic reducer being aware of the semantics of various prevalent languages. In addition, the utilization of LLMs can simplify the customization and extension of reducers, as it would be time-consuming and labor-intensive to manually implement a language-specific reducer or add functionality to an existing one (such as C-Reduce using the Clang frontend to implement C/C++-specific program transformations).

Challenges of Using LLMs. LLMs are not the silver bullet to program reduction. When LLMs are instructed to handle source code as inputs, due to the intrinsic limitations of LLMs [9, 19, 40], LLMs may be unable to understand subtle differences in code [20] and be distracted by irrelevant context [30]. Moreover, in program reduction, input programs typically contain tens of thousands of line of code, which surpasses the input limits of LLMs. Besides, without effective guidance, LLMs are unclear about what transformations to perform.

LLMs-Aided Program Reduction (LPR). We propose LPR in this paper, which is, to the best of our knowledge, the first approach that integrates LLMs for program reduction tasks. LPR synergistically leverages the strengths of both language-generic program reducers and LLMs. Specifically, LPR alternates between invoking a language-generic reducer (we use Perses in experiments) and the LLM. Initially, Perses efficiently reduces large programs to a size manageable for the LLM. Subsequently, the LLM further transforms Perses's output based on specific user-defined prompts that dictate the required transformations. Following this, Perses is re-invoked, as transformations made by the LLM often create additional opportunities for reduction. This process iterates until the program cannot be further minimized. For transformations, we have identified five language-generic transformations to enable further reduction: *Function Inlining*, *Loop Unrolling*, *Data Type Elimination*, *Data Type Simplification*, and *Variable Elimination*.

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

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

Contribution. This study makes the following contributions.

- We introduce LPR, the first attempt to use LLMs for program reduction. By synergizing the capabilities of both language-generic program reducers and LLMs, LPR exhibits both generality across various languages and awareness of semantics in specific languages. Moreover, LPR is easy and flexible to extend with new transformations by simply designing new prompts.
- We propose a multi-level prompting approach to guide LLMs to execute program transformations, and demonstrate its effectiveness in practice. We propose five general-purposed transformations for LLMs to reduce programs or create more reduction opportunities for language-generic program reducers to exploit.
- We comprehensively evaluated LPR on 50 benchmarks across three commonly used languages: C, Rust and JavaScript. Results demonstrate LPR's strong effectiveness and generality.

2 BACKGROUND

2.1 Program Reduction

Given a program P with a certain property, *e.g.*, triggering a compiler bug, the goal of program reduction is to search for a minimal program P_{min} that still triggers the bug. Program reduction has demonstrated its significant usefulness in removing bug-irrelevant code snippets. The original bug-triggering code [17, 23, 31, 49] may have thousands of lines, whereas the distilled version from program reduction tools only contains dozens of lines of code [32]. Some algorithms are designed to generalize across multiple programming languages, while others are customized for certain languages.

2.1.1 Language-Generic Reducers. Some reducers can be generalized to multiple languages. For instance, given the formal syntax of a programming language, algorithms like HDD and Perses can be used to reduce programs corresponding to this language. HDD parses the program into a parse tree and then applies the DDMin [50] at each level of the tree to remove unnecessary tree nodes as much as possible. Perses goes further than HDD by performing certain transformations on the formal syntax to avoid generating syntactically incorrect program variants. Vulcan extends Perses, by introducing novel auxiliary reducers to exhaustively search for smaller variants by replacing identifiers/sub-trees and deleting local combinations of tree nodes on the parse tree.

However, different languages possess their own unique semantic features. Although the aforementioned algorithms are relatively efficient [33], they are incapable of utilizing unique semantics of a particular language to further reduce a program. For example, these algorithms lacks the ability to perform transformations like function inlining. Although Vulcan can identify more reduction opportunities through transformations such as identifier replacement and local exhaustive search, its approach is akin to "brute force" enumeration. This method lacks awareness of the given program's semantics, making it less effective and efficient overall.

2.1.2 Language-Specific Reducers. Previous work has introduced reducers customized for some specific languages. For example, C-Reduce [27] is the most effective reduction tool for C code. It comprises multiple passes that transform the program based on features of the language, thereby making it smaller. Language-specific reducers often rely on static program analysis tools for analysis and modification, *e.g.*, LibTooling [24] is employed in C-Reduce.

However, developing a language-specific reducer is nontrivial. To the best of our knowledge, only a few languages have specific reducers, such as C [27], Java [15, 16], and SMT-LIBv2 [26]. The reason is that the process of designing a reducer for a language, or adding new transformations to an existing reducer, is time-consuming and labor-intensive. For instance, in version 2.10.0 of C-Reduce [28], function inlining was implemented with 604 lines of C++ code. Such challenges impede the development and maintenance of language-specific reducers.

2.2 Large Language Models

Large Language Models refer to a type of deep learning models trained on huge data sets for diverse tasks. The advent of LLMs has opened up numerous potential opportunities across diverse research fields. LLMs are not only proficient in processing natural languages but also exhibit substantial capabilities in understanding and processing programming languages. This highlights the promising future and evolutionary prospects in the realm of software engineering. Recently, LLMs have been applied and assessed on various software engineering tasks, such as automatic program repair [10, 13, 45–47] and program generation [21, 35, 53].

However, despite the usefulness of LLMs, some researchers [20] illustrate that current LLMs are weak in distinguishing nuances between programs. Moreover, the memorizing and processing capacity of LLMs deteriorates as the input size grows [9, 29]. Besides, one cannot expect LLMs to automatically complete complex tasks; they must be guided accordingly [20, 46]. Therefore, for program reduction, directly asking LLMs to reduce programs with tens of thousands of lines is impractical.

3 APPROACH

In this section, we first introduce a motivating example, and then we provide an overview of the LPR workflow. We also outline the details of prompts and proposed transformations in the workflow that enable the LLM to function effectively on given programs.

3.1 Motivation

A motivating example is displayed in Figure 1, where the original code contains highly nested loops, shown in Figure 1a. From Figure 1b to Figure 1c, the nested loops are fully unrolled into hundreds of lines via *Loop Unrolling*, based on the semantic transformations from the LLM. Despite the temporary size increase, the following Perses effectively eliminates all lines except for the bug-relevant one. This is also the final result of LPR, presented in Figure 1d. By contrast, as shown in Figure 1e, Vulcan is incapable of escaping the local minima by exhaustively replacing identifiers and tree nodes. Similarly, C-Reduce cannot fully break down the loop structures, given that it is not integrated with transformations to unroll loops. Even though loop unrolling techniques can be added into C-Reduce in future versions, it will be labor-intensive to implement a specific transformation compared to user-defined prompts in natural language.

3.2 Workflow

Algorithm 1: LPR $(P, \psi, prompts)$

Input: P: the program to be reduced.
Input: $\psi : \mathbb{P} \to \mathbb{B}$: the property to be preserved by P.
Output: P_{min}: the reduced program that preserves the property.

```
P ← Perses (P, ψ) // Quickly minimize P for LLM to process.
repeat /* Monotonically minimize the size of P. */
    P_min ← P
    transformList ← getTransformList (prompts)
    // Iterate through each transformation.
    foreach transform ∈ transformList do
        primaryQuestion ← getPrimaryQuestion (transform)
        followupQuestion ← getFollowupQuestion (transform)
        // Ask LLM to identify a list of targets.
        targetList ← getTargetList(P, primaryQuestion)
        foreach target ∈ targetList do
            // Let LLM apply the transformation on the target.
            P_tmp ← applyTransformation (P, followupQuestion, target)
            if ψ (P_tmp) then
                P ← P_tmp    // P_tmp preserves the property.
        // Run Perses for further reduction.
        P ← Perses (P, ψ)
until |P| ≥ |P_min|
return P_min
```

The overview of the workflow is outlined in Figure 2. Given a bug-triggering program as input, LPR invokes a language-generic reducer and the LLM alternately, until the target program cannot be further reduced. In each iteration, the language-generic reducer efficiently reduces the given program to a minimal one. By contrast, the LLM leverages the semantic knowledge of the language and transforms the program given by the language-generic reducer. This process is guided by user-defined prompts, aiming to expose more reduction potentials to the language-generic reducer.

Algorithm 1 shows LPR's reduction algorithm. Given as inputs (1) a program P targeted for reduction, (2) a property ψ that must be preserved, and (3) the pre-defined *prompts*, LPR generates a reduced

```
......
// nested loop
for (i = 0; i < 7; i++)
  for (j = 0; j < 5; j++)
    for (k = 0; k < 7; k++)
      fn8(ad[i][j][k], "g_643[i][j][k]", aj);
......
```

(a) Original

```
......
// nested loop
for (i = 0; i < 7; i++)
  for (j = 0; j < 5; j++)
    for (k = 0; k < 7; k++)
      s = s ^ ad[i][j][k];
......
```

(b) LPR: Before *Loop Unrolling*

```
......
// the nested loop is fully unrolled
// into hundreds of lines
s = s ^ ad[2][0][5];
s = s ^ ad[2][0][6];
s = s ^ ad[2][1][0];
s = s ^ ad[2][1][1];
......
```

(c) LPR: After *Loop Unrolling*

```
......
// all lines except for the bug-triggering one
// is removed by Perses
s = ad[2][1][0];
......
```

(d) Final result of LPR

```
......
for (i = 0; i < 7; i++)
  for (j = 0; j < 5; j++)
    for (k = 0; k < 7; k++)
      fn8(ad[i][j][k], "g_643[i][j][k]", aj);
......
```

(e) Final result of Vulcan

```
......
for (; h < 7; h++) {
  j = 0;
  for (; j < 5; j++)
    printf("%d\n", c[h][j][0]);
}
......
```

(f) Final result of C-Reduce

Figure 1: Code snippet from LLVM-31259, showcasing the original code, the effectiveness of *Loop Unrolling*, and the final results by LPR, Vulcan and C-Reduce.

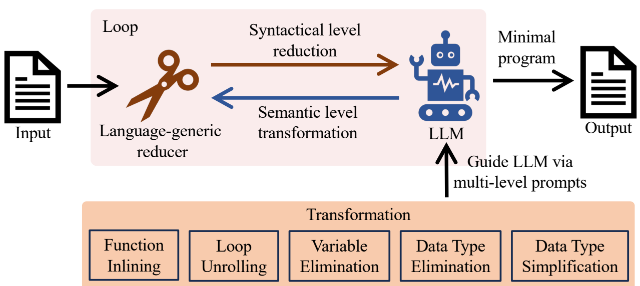


Figure 2: The workflow of LPR.

program P_{min}. The pre-defined *prompts* consist of *primaryQuestion* and *followupQuestion*, in which *primaryQuestion* instructs the LLM to identify a list of targets to be transformed, and *followupQuestion* guides the LLM to apply transformation on each target individually. They will be further introduced in §3.4. Initially, LPR calls Perses to quickly minimize P, so that the size of the program becomes manageable for the LLM to process. Next, LPR loads a sequence of transformations as delineated in line 4, and then iterates through each transformation, as detailed from line 5 to line 13. §3.3 displays the details of each transformation.

During this process, for each transformation, the algorithm retrieves a predefined primary question along with a follow-up question on line 6 – line 7. LPR firstly asks the LLM the primary question under the current program. This query aims to guide the LLM to generate a list of specific targets upon which the transformation will be executed. For instance, for *Loop Unrolling* in the motivating example Figure 1b, the LLM is asked to identify a list of loops in the given program to be unrolled in the *primaryQuestion*, and returns a target list [`for (i = 0; ...)`, `for ( j = 0; ...)`, `for ( k = 0; ...)`].

On line 9 to 12, LPR uses *followupQuestion* to guide the LLM to apply the transformation on each identified target within the program. In the motivating example, the *followupQuestion* can be framed as "Given the program { PROGRAM } and the loop `for(i=0;...)`, optimize it via loop unrolling". The modified program is then extracted from the LLM's response text on line 10. In the example, all loops are unrolled into repeated lines of code in Figure 1c.

In experimental scenarios, given an input program and the prompt, the LLM may generate multiple transformed programs, as the number of responses can be customized. Among all transformed programs, LPR keeps the smallest one that still passes the property test, and discards others. If no transformed program returned from this query satisfies the property, LPR keeps the original one before this query. After each transformation is completed, a language-generic reducer such as Perses [33] is employed to seek additional reduction opportunities, considering that the transformation might have introduced new potential for further simplification. The algorithm persists in the outermost loop until it reaches a fixpoint, signifying that the program size can no longer be reduced.

3.3 Transformations

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

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

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

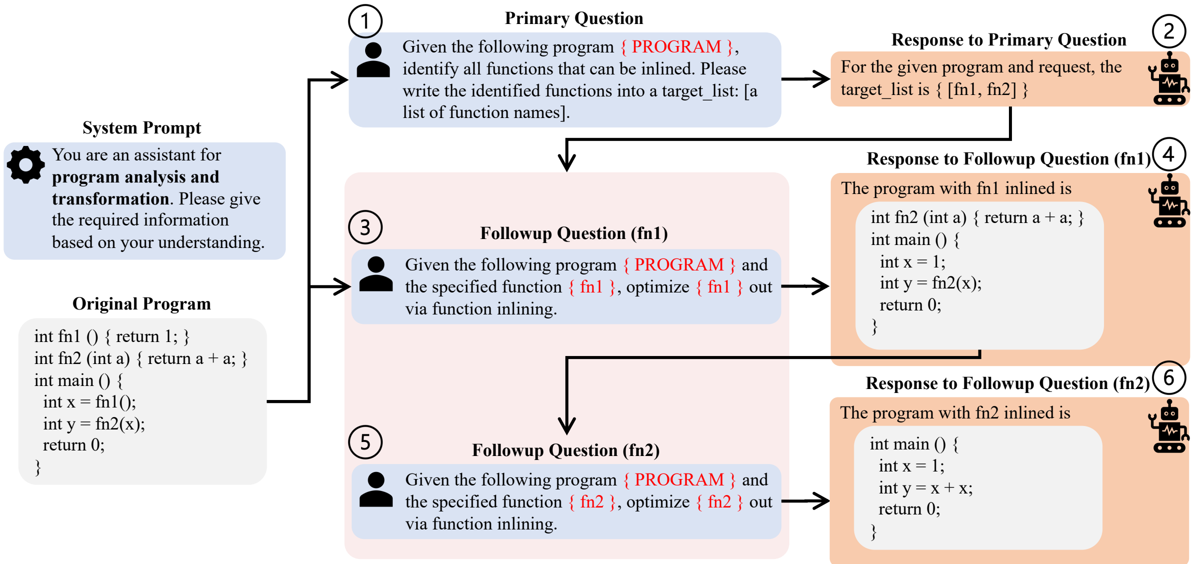


Figure 3: An example of prompt design. and denote system prompt and user prompt provided by the users. denotes the responses from the LLM.

loop structure, while they may be able to reduce the repeated code after loop unrolling, as shown in Figure 1.

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

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

Variable Elimination. Intermediate variables are pervasive in programs, and reducing them is desirable in program reduction tasks. Besides, some variables, although not being used, are hard to eliminate. For instance, to remove an unused parameter, both the parameter defined in the function and its corresponding argument passed to the call site of this function should be removed simultaneously. This is hard or even impossible for language-generic reduction tools. Therefore, we propose *Variable Elimination* to optimize out both intermediate and unused variables.

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

3.4 Multi-level Prompts

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

4 EVALUATION

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

RQ1 What is the effectiveness of LPR in program reduction?

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

RQ3 What is the effectiveness of each transformation in LPR?

4.1 Experimental Setup

Within the workflow of LPR, we employ Perses [33] as the language-agnostic reducer due to its superior efficiency compared to Vulcan. Additionally, we utilize OpenAI API [1], specifically the gpt-3.5-turbo-0613 version, to serve as the LLM. We also develop a variant named LPR+Vulcan, which invokes Vulcan to further reduce the program after LPR finishes. The experiments are conducted on an Ubuntu 22 server with an Intel(R) Xeon(R) Gold 6348 CPU @ 2.60GHz and 512 GB RAM. For a fair comparison, all algorithms were executed in a single-process, single-thread environment.

Benchmarks. To measure the effectiveness and efficiency of LPR across various languages, we employ three benchmark suites: Benchmark-C, Benchmark-Rust and Benchmark-JS. The Benchmark-C, previously collected and utilized by previous studies [38, 48, 52], comprises 20 large complex programs triggering real-world bugs in LLVM or GCC. Benchmark-Rust, incorporating 20 bug-triggering Rust programs, has also been used in prior research [48]. We further craft Benchmark-JS, a non-public benchmark suite, for this study. Specifically, we use FuzzJIT [41] to fuzz a prevalent JavaScript engine, *i.e.*, JavaScriptCore (version c6a5bcc), and then randomly collect 10 programs that cause miscompilations in JIT compiler. Since the programs and reduced programs in Benchmark-JS are not publicly accessible at the time of LLMs' training, and thus not in the training sets of LLMs. The evaluation on Benchmark-JS helps to investigate whether LPR suffers from the data leakage problem [44]. In total, the evaluation benchmarks encompass 50 programs triggering real-world bugs in compilers, spanning across three popular programming languages.

Baselines. In all three benchmark suites, we use Perses and Vulcan as baselines. Perses stands out as a highly effective and efficient program reduction tool. To avoid the occurrence of syntactical invalid variants during the reduction process, it transforms and normalizes the formal syntax of a programming language. Vulcan [48], building upon Perses, provides three manually designed auxiliary reducers to further search for reduction opportunities on results from Perses. Compared to Perses, Vulcan achieves a reduction in the number of tokens, albeit at the expense of increased running time. They are both language-generic and are applicable across a broad spectrum of programming languages. We also include C-Reduce (v2.9.0) as an additional baseline. C-Reduce not only stands as the most effective algorithm for C, it can also be applied to other languages, though not customized for them.

Configuration. If not otherwise specified, our experiments are conducted by invoking OpenAI API (version gpt-3.5-turbo-0613), with the proposed multi-level prompt and transformations in §3.3 and §3.4. To effectively harness the inherent randomness of LLMs, we set `temperature=1.0`. This high value encourages the LLM to generate more diverse outcomes [3]. Additionally, we employ `n=5` to generate five distinct results for every query [2], enabling us to choose the smallest program preserving the property ψ as the optimal result. All the rest configurations are set to their default values.

4.2 RQ1: Reduction Effectiveness

We measure the effectiveness of LPR, LPR+Vulcan and baseline algorithms via the final program sizes in tokens. A smaller size is favored, as it signifies the removal of more bug-irrelevant code, thereby saving developers more manual effort. The effectiveness of each algorithm on all three benchmark suites is presented in Table 1. Due to the randomness of LPR, we repeat five times for LPR and LPR+Vulcan on every benchmark, and display the mean value and standard deviation value in the table. On each benchmark, the minimal results are highlighted in bold.

Table 1: The reduction sizes of Perses, Vulcan, C-Reduce, LPR and LPR+Vulcan. Best results among all algorithms are highlighted in bold font.

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

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

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

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

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

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

4.3 RQ2: Reduction Efficiency

In this research question, we measure the time elapsed when users employ each technique for program reduction.[1] Shorter time indicates higher efficiency, and Table 2 shows the results. Since both Vulcan and LPR perform reduction on top of Perses's results, it is impossible for these two algorithms to take less time than Perses. In addition, as a highly efficient tree-based reduction algorithm, Perses is generally faster than C-Reduce. Therefore, we focus on the comparison among Vulcan, C-Reduce and LPR.

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

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

After further in-depth analysis, another interesting fact emerges. On the three benchmark suites, the average time taken by LPR is 1.915, 1.839, and 0.174 hours, respectively. However, within these durations, the time spent waiting for the LLM responses accounts

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

Table 2: The reduction time (in the format of hh:mm:ss) of Perses, Vulcan, C-Reduce, LPR and LPR+Vulcan.

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

for 32.26%, 28.11%, and 72.99%. Additionally, we measure the efficiency from more perspectives. Take Benchmark-C as an example: the average expense to finish each benchmark amounts to \$0.42, with each benchmark requiring 41 queries on average, and every query consuming approximately 39 seconds on average. This experiment involves invoking the OpenAI API, and might have been limited by high user traffic and few computational resources allocated. Considering the ongoing advancements in LLMs technology, we believe that LPR's efficiency will be substantially improved in the future.

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

4.4 RQ3: Effectiveness of Each Transformation

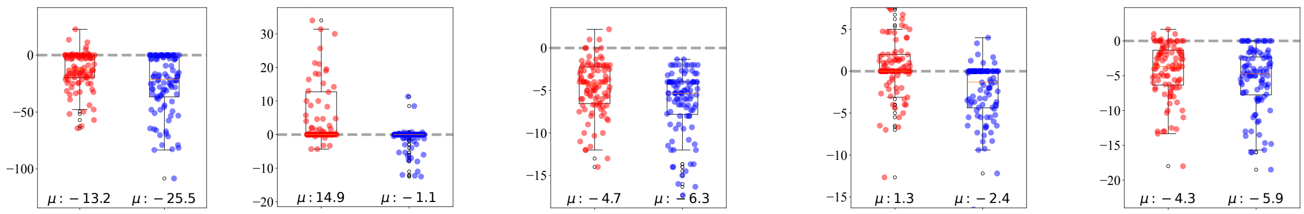
To answer this question, we delve into the impact of each transformation on program size, alongside their potential to escape the local minimal program and unlock new reduction opportunities. Our in-depth analysis focuses on Benchmark-C because of its complexity of bug-triggering programs compared to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

For each transformation, we measure how the program size changes in each benchmark of Benchmark-C after a specific transformation is performed, and plot the size changes into a box-plot and red dots, as shown in Figure 4. Additionally, since each transformation is immediately followed by an invocation of Perses, we also monitor the cumulative size changes resulting from both the transformation and its subsequent Perses reduction. These changes are depicted by the blue dots in the second box-plot of each subplot in Figure 4. Given that a transformation is performed in every iteration, we compute the average size change by dividing the total sum of size changes for that transformation by the number of iterations. This approach enables us to thoroughly comprehend how each transformation influences program size and assess its capacity to provide further reduction opportunities to Perses.

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

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

To further understand the impact of each transformation on the entirety of Benchmark-C, we provide a detailed analysis of their contributions. Specifically, on the 5 repeated experiments on 20 benchmarks, we calculate the average size reduction brought about by each transformation across all these 100 runs by summing up the size decreases attributed to the transformation in each benchmark and then computing the mean value, as illustrated in Figure 5.



(a) ***Function Inlining*** **(b)** ***Loop Unrolling*** **(c)** ***Data Type Elimination*** **(d)** ***Data Type Simplification*** **(e)** ***Variable Elimination***

Figure 4: Program size changes induced by each transformation on benchmarks of Benchmark-C. In each subplot, the left box-plot and red dots represent how the size of each program changes before and after executing the transformation. The right box-plot and blue dots represent the size change of each benchmark after executing the transformation and the follow-up Perses reduction. There are a total of 20 benchmarks in Benchmark-C, and each experiment is repeated 5 times. Therefore, we draw 100 data points on each boxplot.

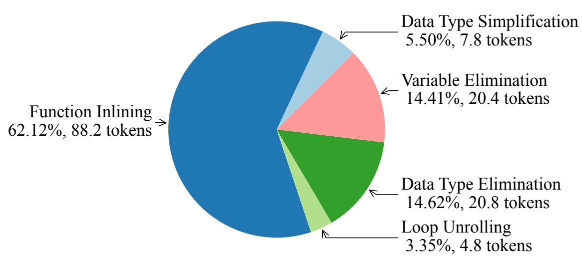


Figure 5: Average size decrease and its percentage induced by each transformation within Benchmark-C.

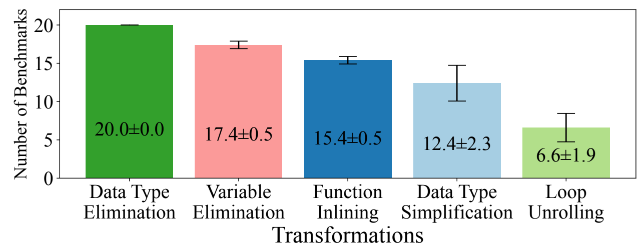


Figure 6: The number of benchmarks impacted by each transformation within Benchmark-C.

Additionally, we quantify the prevalence of each transformation by counting the number of benchmarks in which it induces a size decrease, as demonstrated in Figure 6. The above evaluations together offer a comprehensive view of how each transformation influences program sizes and how frequently they take effect within Benchmark-C.

From Figure 5, it is evident that every transformation contributes to further size reduction in Benchmark-C. Specifically, while *Function Inlining* is responsible for a reduction of 88.2 tokens on average, contributing 62.12% to the overall decrease. *Loop Unrolling* shows a minimal effect, contributing only 4.8 tokens, 3.35% to the overall decrease. This highlights the varying degrees of influence each transformation has on program size. Further insights from Figure 6 reveal that *Data Type Elimination* affects all benchmarks, likely due to the ubiquitous presence of `typedef` across all benchmarks. In contrast, *Loop Unrolling* is the least prevalent transformation, affecting merely 6 on average out of 20 benchmarks. This outcome is expected, considering that not all programs involve loop structures, and not every loop is irrelevant to the compiler bugs. In addition, the relatively higher standard deviations observed in the *Data Type Simplification* and *Loop Unrolling* suggest that these transformations present greater challenges for the LLM to effectively handle.

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

5 DISCUSSION

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

5.1 The Effectiveness of Multi-level Prompt

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

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

5.2 The Impact of Temperature

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

Table 3: The impact of temperature on reduction sizes.

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

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

Given that program reduction is an NP-complete problem, randomness of LLMs has its advantages and drawbacks. Take the results of RQ1 in Table 1 as an example. On the one hand, randomness allows LPR to explore more distinct local minima, and sometimes generates smaller programs than C-Reduce in five repeated experiments on each benchmark. On the other hand, randomness of LLMs introduces variability. While the standard deviation remains below 10 in most benchmarks, it may significantly increase in certain benchmarks, such as > 60 in GCC-59903. This variation can be attributed to the high complexity of the given program. In such scenarios, LLMs might not consistently execute accurate transformations, resulting in a range of local minima and divergent results.

5.3 Failures in LPR

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

Failures indeed lead to more property tests before making any progress, but they are inevitable, as program reduction is a trial-and-error process. Even without LLM, property checks issued by Perses and Vulcan usually have a failure rate of around 90%. As shown in Table 1 and Table 2, LPR is effective and efficient enough, illustrating that failures can be mitigated via multiple responses.

5.4 Threats to Validity

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

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

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

5.4.2 Threats to External Validity. One threat to external validity is the generalizability of LPR across languages. Although the approach of LPR is language-agnostic, LLMs used by LPR may have limited knowledge of certain languages, which may affect the performance of LPR. To mitigate this threat, we evaluate LPR on three prevalent programming languages, namely, C, Rust and JavaScript. The evaluation results demonstrated the generalizability of LPR on diverse popular programming languages. For languages that are not familiar to LLMs, LPR may still produce reasonable results, since it is shown that LLMs like Codex still perform well in less popular languages like Lua (0.2% in Github) [5]. As for a completely new language that LLMs cannot recognize and process, a possible solution is to incorporate the prompts in LPR with few-shot prompting [4, 22], so that LLMs learn how to recognize and process a new language from the given examples. Further exploration of this approach will be left as future work.

An additional threat is the applicability of our approach across different LLMs. To mitigate this threat, on the same benchmarks used in §5.2, we repeat the experiments with CodeLlama, another widely-used LLM family. As illustrated in Table 4, the experimental results on two models of CodeLlama, *i.e.*, CodeLlama-13b and CodeLlama-34b, show no significant differences from results on ChatGPT. Furthermore, as LLMs continue to evolve, we anticipate improvements in both quality and time of code processing.

Table 4: The reduction sizes of LPR using various LLMs.

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

6 RELATED WORK

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

6.1 Program Reduction

DDMin [50] initiated the research topic of program reduction. It treats the input as a list of elements, and consistently splits the list into halves. Then it iteratively attempts to reduce the input list by exploring subsets and their complements at varying levels of granularity, transitioning from coarse to fine. Hierarchical Delta Debugging [25], short for HDD, parses the program input into a parsing tree, and performs DDMin on each level of the tree structure. Perses [33] avoids the generation of syntactically invalid program variants during reduction by formal syntax transformation. Vulcan, further pushes the limit of Perses via identifier/sub-tree replacement and local exhaustive search [48]. RCC is a compact refreshable caching scheme to speed up program reduction [38]. All the above works are not customized for certain languages, though having high generality, they lack semantic knowledge of a certain language for further reduction.

Besides, some tools are specifically designed for certain languages. C-Reduce [27], incorporating various transformation passes for features in C, is the most effective reducer on this language. J-Reduce [15, 16] is a tool for Java bytecode reduction, it reformulates the bytecode reduction into dependency graph simplification. ddSMT [26] is designed for reducing programs in SMT-LIBv2 format. All these works leverage language features to reduce more effectively than language-generic tools.

Distinct from prior work, LPR synergistically combines LLMs and language-generic reduction tools to harness the advantages of both. Language-generic reducers stand out for their remarkable generality across multiple languages, while LLMs excel in further refining the programs with the domain knowledge of certain languages learned from large training sets. Language-specific tools typically demand considerable human effort to design and implement feature-related transformations for reduction, while LPR requires only a few lines of natural language prompts, significantly reducing the effort involved.

6.2 LLMs for Software Engineering

Large Language Models (LLMs) have proved their remarkable capability of undertaking multiple text-processing tasks, including source code-related works. Recent works focus on applying LLMs to facilitate software engineering tasks, or assessing the effectiveness, potential and limitations of LLMs on software development and maintenance. Some research [13, 45–47] focus on empirically applying LLMs on automatic program repair (APR). Huang *et al.* [13] performed an empirical study on improvement brought by model fine-tuning in APR. Xia *et al.* [46] thoroughly evaluated 9 state-of-the-art LLMs across multiple datasets and programming languages, and demonstrated that directly applying LLMs has already significantly outperformed all existing APR techniques. Additionally, some works focus on LLMs' performance *w.r.t.* code completion, generation and fuzzing [6, 21, 35, 53], by leveraging the code analysis and generation ability of LLMs.

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

7 CONCLUSION

This paper proposes LLMs-aided program reduction (LPR), which is the first approach that leverages LLMs for the program reduction task to the best of our knowledge. By combining the strength of LLMs and existing language-generic program reduction techniques, LPR can perform language-specific transformations to effectively reduce the program while being language-generic (*i.e.*, can be easily applied to a wide range of languages). The evaluation shows that in 50 benchmarks across three programming languages. LPR significantly outperforms Vulcan. Specifically, LPR produces 24.93%, 4.47%, and 11.71% smaller programs on C, Rust, and JavaScript, respectively. Meanwhile, The evaluation also demonstrates that LPR complements Vulcan to some extent. By reducing the outputs of LPR with Vulcan, we attained results that have similar sizes to those of C-Reduce in Benchmark-C. In terms of efficiency perceived by users, LPR excels in reducing complex programs, and takes 10.77%, 34.88%, 36.96% less time than Vulcan to finish all the benchmarks, respectively.

8 DATA AVAILABILITY

For replication, we have implemented LPR and released it publicly [51].

ACKNOWLEDGMENTS

We thank all the anonymous reviewers in ISSTA'24 for their insightful feedback and comments. This research is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) through the Discovery Grant, a project under WHJIL, and CFI-JELF Project #40736.

REFERENCES

[1] 2023. *OpenAI API.* Retrieved 2023-11-20 from https://platform.openai.com/docs/overview
[2] 2023. *OpenAI API: N.* Retrieved 2023-11-20 from https://platform.openai.com/docs/api-reference/chat/create#chat-create-n
[3] 2023. *OpenAI API: Temperature.* Retrieved 2023-11-20 from https://platform.openai.com/docs/api-reference/chat/create#chat-create-temperature

[4] Toufique Ahmed and Premkumar Devanbu. 2022. Few-shot training llms for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–5. https://doi.org/10.48550/arXiv.2207.04237

[5] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2022. Multipl-e: A scalable and extensible approach to benchmarking neural code generation. *arXiv preprint arXiv:2208.08227* (2022). https://doi.org/10.48550/arXiv.2208.08227

[6] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*. 423–435. https://doi.org/10.1145/3597926.3598067

[7] Alastair Donaldson and David MacIver. 2021. *Test Case Reduction: Beyond Bugs*. Retrieved May 29, 2023 from https://blog.sigplan.org/2021/05/25/test-case-reduction-beyond-bugs

[8] Alastair F Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpiński. 2021. Test-case reduction and deduplication almost for free with transformation-based compiler testing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1017–1032. https://doi.org/10.1145/3453483.3454092

[9] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 31–53. https://doi.org/10.1109/ICSE-FoSE59343.2023.00008

[10] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1469–1481. https://doi.org/10.1109/ICSE48619.2023.00128

[11] Qiuhan Gu. 2023. LLM-Based Code Generation Method for Golang Compiler Testing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 2201–2203. https://doi.org/10.1145/3611643.3617850

[12] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 380–394. https://doi.org/10.1145/3243734.3243838

[13] Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023. An Empirical Study on Fine-Tuning Large Language Models of Code for Automated Program Repair. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 1162–1174. https://doi.org/10.1109/ASE56229.2023.00181

[14] Naman Jain, Skanda Vaidyanath, Arun Shankar Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram K. Rajamani, and Rahul Sharma. 2022. Jigsaw: Large Language Models meet Program Synthesis. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1219–1231. https://doi.org/10.1145/3510003.3510203

[15] Christian Gram Kalhauge and Jens Palsberg. 2019. Binary reduction of dependency graphs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 556–566. https://doi.org/10.1145/3338906.3338956

[16] Christian Gram Kalhauge and Jens Palsberg. 2021. Logical bytecode reduction. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1003–1016. https://doi.org/10.1145/3453483.3454091

[17] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices* 49, 6 (2014), 216–226. https://doi.org/10.1145/2594291.2594334

[18] Bastien Lecoeur, Hasan Mohsin, and Alastair F. Donaldson. 2023. Program Reconditioning: Avoiding Undefined Behaviour When Finding and Reducing Compiler Bugs. *Proc. ACM Program. Lang.* 7, PLDI, Article 180 (jun 2023), 25 pages. https://doi.org/10.1145/3591294

[19] Junyi Li, Xiaoxue Cheng, Xin Zhao, Jian-Yun Nie, and Ji-Rong Wen. 2023. HaluEval: A Large-Scale Hallucination Evaluation Benchmark for Large Language Models. In *The 2023 Conference on Empirical Methods in Natural Language Processing*. https://doi.org/10.48550/arXiv.2305.11747

[20] Tsz On Li, Wenxi Zong, Yibo Wang, Haoye Tian, Ying Wang, Shing-Chi Cheung, and Jeff Kramer. 2023. Nuances are the Key: Unlocking ChatGPT to Find Failure-Inducing Tests with Differential Prompting. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 14–26. https://doi.org/10.1109/ASE56229.2023.00089

[21] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210* (2023). https://doi.org/10.48550/arXiv.2305.01210

[22] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *Comput. Surveys* 55, 9 (2023), 1–35. https://doi.org/10.1145/3560815

[23] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–25. https://doi.org/10.1145/1993498.1993532

[24] LLVM. 2000. *LibTooling*. https://clang.llvm.org/docs/LibTooling.html Accessed: 2023-04-30.

[25] Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering*. 142–151. https://doi.org/10.1145/1134285.1134307

[26] Aina Niemetz and Armin Biere. 2013. ddSMT: a delta debugger for the SMT-LIB v2 format. In *Proceedings of the 11th International Workshop on Satisfiability Modulo Theories, SMT*. 8–9.

[27] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 335–346. https://doi.org/10.1145/2254064.2254104

[28] John et al. Regehr. 2012. *C-Reduce*. Retrieved 2023-11-26 from https://github.com/csmith-project/creduce

[29] Uri Shaham, Elad Segal, Maor Ivgi, Avia Efrat, Ori Yoran, Adi Haviv, Ankit Gupta, Wenhan Xiong, Mor Geva, Jonathan Berant, and Omer Levy. 2022. SCROLLS: Standardized CompaRison Over Long Language Sequences. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (Eds.). Association for Computational Linguistics, 12007–12021. https://doi.org/10.18653/V1/2022.EMNLP-MAIN.823

[30] Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H Chi, Nathanael Schärli, and Denny Zhou. 2023. Large language models can be easily distracted by irrelevant context. In *International Conference on Machine Learning*. PMLR, 31210–31227. https://doi.org/10.5555/3618408.3619699

[31] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 849–863. https://doi.org/10.1145/2983990.2984038

[32] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 294–305. https://doi.org/10.1145/2931037.2931074

[33] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering*. 361–371. https://doi.org/10.1145/3180155.3180236

[34] Maolin Sun, Yibiao Yang, Yang Wang, Ming Wen, Haoxiang Jia, and Yuming Zhou. 2023. SMT Solver Validation Empowered by Large Pre-Trained Language Models. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 1288–1300. https://doi.org/10.1109/ASE56229.2023.00180

[35] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. 2023. Is ChatGPT the Ultimate Programming Assistant–How far is it? *arXiv preprint arXiv:2304.11938* (2023). https://doi.org/10.48550/arXiv.2304.11938

[36] Jia Le Tian, Mengxiao Zhang, Zhenyang Xu, Yongqiang Tian, Yiwen Dong, and Chengnian Sun. 2023. Ad Hoc Syntax-Guided Program Reduction. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 2137–2141. https://doi.org/10.1145/3611643.3613101

[37] Yongqiang Tian, Zhenyang Xu, Yiwen Dong, Chengnian Sun, and Shing-Chi Cheung. 2023. Revisiting the Evaluation of Deep Learning-Based Compiler Testing. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, 19th-25th August 2023, Macao, SAR, China*. ijcai.org, 4873–4882. https://doi.org/10.24963/IJCAI.2023/542

[38] Yongqiang Tian, Xueyan Zhang, Yiwen Dong, Zhenyang Xu, Mengxiao Zhang, Yu Jiang, Shing-Chi Cheung, and Chengnian Sun. 2023. On the Caching Schemes to Speed Up Program Reduction. *ACM Trans. Softw. Eng. Methodol.* 33, 1, Article 17 (nov 2023), 30 pages. https://doi.org/10.1145/3617172

[39] Guancheng Wang, Ruobing Shen, Junjie Chen, Yingfei Xiong, and Lu Zhang. 2021. Probabilistic Delta debugging. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 881–892. https://doi.org/10.1145/3468264.3468625

[40] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software Testing with Large Language Models: Survey, Landscape, and Vision. https://doi.org/10.48550/arXiv.2307.07221 arXiv:2307.07221 [cs.SE]

[41] Junjie Wang, Zhiyi Zhang, Shuang Liu, Xiaoning Du, and Junjie Chen. 2023. FuzzJIT: Oracle-Enhanced Fuzzing for JavaScript Engine JIT Compiler. In *USENIX Security Symposium. USENIX.*

[42] Theodore Luo Wang, Yongqiang Tian, Yiwen Dong, Zhenyang Xu, and Chengnian Sun. 2023. Compilation Consistency Modulo Debug Information. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 146–158. https://doi.org/10.1145/3575693.3575740

[43] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 172–184. https://doi.org/10.1145/3611643.3616271

[44] Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. 2023. How Effective Are Neural Networks for Fixing Security Vulnerabilities. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (<conf-loc>, <city>Seattle</city>, <state>WA</state>, <country>USA</country>, </conf-loc>) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1282–1294. https://doi.org/10.1145/3597926.3598135

[45] Chunqiu Steven Xia, Yifeng Ding, and Lingming Zhang. 2023. Revisiting the Plastic Surgery Hypothesis via Large Language Models. *arXiv preprint arXiv:2303.10494* (2023). https://doi.org/10.48550/ARXIV.2303.10494

[46] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery.* https://doi.org/10.1109/ICSE48619.2023.00129

[47] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. *arXiv preprint arXiv:2304.00385* (2023). https://doi.org/10.48550/arXiv.2304.00385

[48] Zhenyang Xu, Yongqiang Tian, Mengxiao Zhang, Gaosen Zhao, Yu Jiang, and Chengnian Sun. 2023. Pushing the Limit of 1-Minimality of Language-Agnostic Program Reduction. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 636–664. https://doi.org/10.1145/3586049

[49] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation.* 283–294. https://doi.org/10.1145/1993316.1993532

[50] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200. https://doi.org/10.1109/32.988498

[51] Mengxiao Zhang, Yongqiang Tian, Zhenyang Xu, Yiwen Dong, Shin Hwei Tan, and Chengnian Sun. 2024. Artifact for "LPR: Large Language Models-Aided Program Reduction". https://doi.org/10.5281/zenodo.10901714

[52] Mengxiao Zhang, Zhenyang Xu, Yongqiang Tian, Yu Jiang, and Chengnian Sun. 2023. PPR: Pairwise Program Reduction. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 338–349.

[53] Li Zhong and Zilong Wang. 2023. A study on robustness and reliability of large language model code generation. *arXiv preprint arXiv:2308.10335* (2023). https://doi.org/10.48550/arXiv.2308.10335

Received 16-DEC-2023; accepted 2024-03-02