PPR: Pairwise Program Reduction

Mengxiao Zhang
m492zhan@uwaterloo.ca
University of Waterloo
Canada

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

Yongqiang Tian
yongqiang.tian@uwaterloo.ca
University of Waterloo
Canada

Yu Jiang
jy1989@mail.tsinghua.edu.cn
Tsinghua University
China

Chengnian Sun
cnsun@uwaterloo.ca
University of Waterloo
Canada

ABSTRACT

Program reduction is a practical technique widely used for debugging compilers. To report a compiler bug with a bug-triggering program, one needs to minimize the program by removing bug-irrelevant program elements first. Though existing program reduction techniques, such as C-Reduce and Perses, can reduce a bug-triggering program as a whole, they overlook the fact that the degree of relevance of each remaining token to the bug varies.

To this end, we propose Pairwise Program Reduction (PPR), a new program reduction technique for minimizing a pair of programs *w.r.t.* certain properties. Given a seed program P_s, a variant P_v derived from P_s, and the properties P_s and P_v exhibit separately (*e.g.*, P_v crashes a compiler whereas P_s does not), PPR not only reduces the sizes of P_s and P_v, but also minimizes the differences between P_s and P_v. The final result of PPR is a pair of minimized programs that still preserve the properties, but the minimized differences between the pair highlight the critical program elements that are highly related to the bug.

To thoroughly evaluate PPR, we manually constructed the first pairwise benchmark suite from real-world compiler bugs (20 bugs in GCC and LLVM, 9 bugs in Rustc and 9 bugs in JerryScript). The evaluation results show that PPR significantly outperforms the baseline: DD, a variant of Delta Debugging. Specifically, on large and complex programs, PPR's reduction results are only 0.6% of those by DD *w.r.t.* program size. The sizes of the minimized variants (*i.e.*, P_v) by PPR are also comparable to those by Perses and C-Reduce; but PPR offers more for debugging by highlighting the critical, bug-inducing changes via the minimized differences. Evaluation on Rust and JavaScript demonstrates PPR's strong generality to other languages.

CCS CONCEPTS

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

KEYWORDS

Program Reduction, Delta Debugging, Bug Isolation

ACM Reference Format:
Mengxiao Zhang, Zhenyang Xu, Yongqiang Tian, Yu Jiang, and Chengnian Sun. 2023. PPR: Pairwise Program Reduction. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3611643.3616275

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616275

1 INTRODUCTION

Compilers are fundamental system software to build all software including compilers themselves; any bug in compilers may incur significant, profound damages. Thus it is crucial to detect and fix compiler bugs promptly. In recent years, researchers have devoted dedicated efforts to random compiler testing [2, 10, 18, 22–25, 29, 31, 32, 36, 47]. Among all compiler testing techniques, one major, widely used type of techniques is *mutation-based* testing [2, 10, 18, 22–24, 32, 36, 42], which generates new test programs (referred to as *variants*) by applying random mutations to existing programs (referred to as *seeds*), in hopes of provoking compiler bugs.

Program Reduction. Upon finding a program P that triggers a compiler bug, program reduction—an important and highly demanded category of techniques for compiler validation and debugging [15, 17, 21, 26, 28, 35, 37, 40, 48]—aims to minimize P by removing code elements that are irrelevant to compiler bugs; the minimized program is used to facilitate diagnosing and fixing bugs.

Most reduction techniques reduce a program as a whole and assume that every part of the minimized program is equally relevant to the bug, but overlook the fact that the degree of relevance of each program element to the bug varies. For instance, though the state-of-the-art technique C-Reduce can minimize the original bug-triggering program of LLVM-21467 [5] to 103 tokens as shown in Figure 1a, it can still take compiler developers large efforts to debug, as the critical bug-triggering tokens are not highlighted. In this example, certain tokens are indeed more relevant to the bug than the others: the bug is triggered by code highlighted in Figure 1c, which modifies the same variable `b` in multiple basic blocks and is more relevant to the bug than statements such as function calls to `div` and `id`.

```c
#include <stdint.h>
uint8_t g;
int16_t (div)(int16_t a, int16_t b) {
  return b == 0 ?: a / b;
}
int64_t (id)() { return 0; }
int8_t fn(a) {
  int32_t d = 1;
  uint8_t b;
  for (;;) {
    uint16_t c = id(b |= g);
    if (div(c,
      div(10, 65535)))
      b |= g;
    else
      d &= a;
    b |= g;
  }
}

int main() { fn(0); }
```

(a) By C-Reduce.

```c
#include <stdint.h>
uint8_t g;
int16_t div(int16_t a, int16_t b) {
  return b == 0 ? a : a / b;
}
int64_t id(a) { return a; }
int8_t fn(a) {
  uint8_t b;
  for (;;) {
    uint16_t c = 65532UL;
    id(b |= g);
    if (div(c,
      div(0x2FAAL, 65535UL)))


      a &= 1L;


  }
}
int main() { fn(0); }
```

(b) *Seed* by PPR.

```c
#include <stdint.h>
uint8_t g;
int16_t div(int16_t a, int16_t b) {
  return b == 0 ? a : a / b;
}
int64_t id(a) { return a; }
int8_t fn(a) {
  uint8_t b;
  for (;;) {
    uint16_t c = 65532UL;
    id(b |= g);
    if (div(c,
      div(0x2FAAL, 65535UL)))
      b |= g;
    else {
      a &= 1L;
    }
    b |= g;
  }
}
int main() { fn(0); }
```

(c) *Variant* by PPR.

```c
......
int8_t fn() {
  ......
}
......
int main() {
  ......
  fn;
}
```

(d) *Seed* by DD.

```c
......
int8_t fn() {
  ......
}
......
int main() {
  ......
  fn();
}
```

(e) *Variant* by DD.

Figure 1: Reduction results by C-Reduce (a), PPR (b and c) and DD (d and e) for LLVM-21467. The differences between the seed and variant are highlighted in blue. All programs are slightly adjusted and aligned for clarity. We simplify results from DD for illustration due to the large size of the original version.

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

DD DD [48] is a variant in the Delta Debugging algorithm family.[1] Given P, it does not only reduce P but also highlights the key failure-inducing elements in P. It returns a pair of programs: the minimized bug-triggering program (referred to as *variant*) reduced from P and the maximized non-bug-triggering program (referred to as *seed*) grown from an empty program. However, DD does not respect the syntactical constraints of programs and thus does not perform well on programs: the reduction time can be excessively long because of generating large numbers of syntactically invalid programs and the results are usually too large for debugging [28, 37]. Moreover, DD does not use the original seed program, from which P is derive, to find the failure-inducing program elements, and thus the difference between the seed and variant is usually irrelevant to the root cause of the bug. Figures 1d and 1e show the reduced result by DD for LLVM-21467: DD highlights that if the function call `fn` is removed from the variant, then the bug disappears. Though the bug is indeed triggered by certain code in `fn`, the highlighted difference by DD is far less informative than the difference in Figure 1c. Moreover, in the reduced variant by DD, `fn` has 4332 tokens, which contains excessive irrelevant information and thus complicates debugging (more details are in §2).

Spirv-fuzz Spirv-fuzz [12] is a testing technique for SPIR-V intermediate representation [44]. Given a seed program P_s, it applies a sequence of random mutations on P_s to generate a variant for testing SPIR-V processors. Different from classical mutation-based testing, Spirv-fuzz intentionally makes each mutation as small and independent as possible. Once a variant P_v finds a bug, DDMin is used to reduce the sequence of mutations that is applied on P_s to generate P_v, so that the bug-irrelevant mutations can be removed; in the end, P_v and the sequence of bug-relevant mutations are included in the bug report to facilitate debugging.

Although Spirv-fuzz can highlight the critical changes and has been adopted at Google [11], several weaknesses limit its application to testing other language implementations, and motivate this work. First, Spirv-fuzz relies on a special mutation-base technique: the mutation operations need to be small and independent, and the sequence of mutations applied to a seed needs to be recorded, modifiable, and replayed. Second, it can merely minimize the differences between the seed and variant, but is not capable of reducing the common part: The authors also acknowledged this limitation and that large seed programs might diminish the comprehensibility of the bug reports, as quoted from [12]: "The comprehensibility of these bug reports and regression tests depends on a reasonably small original program".

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

[1] Zeller *et al.* proposed two algorithms DD and DDMin in [48]. DD aims to isolate failure-inducing changes by narrowing the difference between a given list that has a property and an empty list; DDMin removes from the given list the elements that are irrelevant to the property.

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

For Developers of Compilers. Debugging compiler bugs is arduous, especially debugging miscompilation faults.[2] The minimized differences between P_s^* and P_v^* highlight the minimal required changes from a passing program to a bug-triggering one to induce the bug. Such minimal changes, together with the minimized bug-triggering program, offer a good starting point to compiler developers for fault localization and analysis.

For Researchers on Mutation-Based Compiler Testing. Isolating the critical changes provides insights to researchers on compiler testing, who frequently need to analyze the efficacy of mutations (*e.g.*, which mutation operator is more effective in triggering compiler bugs) to understand and tune their fuzzing techniques. Prior mutation-based research work indeed conducted such analysis, though manually. For example, researchers distilled and posted critical code mutations that contributed to a bug [2, 36, 42] in their paper. However, such manual efforts are often labor-intensive and error-prone, and more importantly can be automated.

We have conducted extensive evaluations of PPR, showcasing its practicality and superiority over DD. First, we evaluated PPR on 20 large real-world C compiler bugs. On average, the number of tokens involved in bug-triggering changes is reduced from 27,880 to only 24, and the overall input size is narrowed from 52,712 to 278 tokens. This result is remarkably small compared to 47,850 tokens from DD, and even comparable to results of the state-of-the-art reduction techniques, *i.e.*, Perses [37] and C-Reduce [35]. For efficiency, PPR is faster than DD on average. Besides C/C++ programs, further evaluation shows PPR's generality to other languages.

Contributions. We make the following major contributions.

- We propose the novel, general concept of Pairwise Program Reduction for program reduction and failure diagnosis. It complements classical program reduction approaches by identifying critical bug-triggering differences.
- We propose the first algorithm for pairwise program reduction, including three effective and general reducers to reduce both differences and commonalities between a pair of programs.
- To comprehensively evaluate the performance of PPR, we designed benchmarks ranging from small programs via simple mutations to large programs via complex fuzzing techniques.
- To enable future research on this line of research, we released the source code of PPR and the benchmarks publicly [49].

2 MOTIVATION

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

LLVM-21467 is a bug in LLVM-3.4.2. The original bug-triggering variant contains 28,012 tokens in total, and provokes an infinite loop when being compiled by Clang-3.4.2 with the `-O2` flag. As explained in the bug report [27], the optimization pass `FoldOpIntoPhi` folds certain instructions into PHI nodes, for intermediate analysis in SSA form [45]. When processing a variable in a basic block, it tries to invoke itself if the next reachable block has the same variable. When `FoldOpIntoPhi` encounters two mutually reachable basic blocks modifying the same variable, like `b |= g` on line 14 and line 18 in Figure 1c, an infinite recursion occurs.

Classical Program Reduction (C-Reduce). In this example, the variant is reduced to 103 tokens by C-Reduce shown in Figure 1a. While this outcome seems favorable at first glance—with over 99% of the code removed—there still remain slightly more than a hundred tokens. Consequently, it becomes challenging for developers to determine which part of the code is more suspicious than the rest.

Pairwise Program Reduction. By contrast, PPR not only minimizes bug-triggering program in Figure 1c, but also provides its corresponding, minimized seed for comparison in Figure 1b, as well as a minimal set of changes. In this case, reduced seed/variant has 89/100 tokens, comparable to the result by C-Reduce *w.r.t.* size. More importantly, the reduced differences reveal that the change of control flow and multiple assignments to the same variable are relevant to the bug, as highlighted in Figure 1c. PPR detects all critical changes successfully, offering developers with a better starting point for fault diagnosis.

DD. Results from DD in Figure 1d and in Figure 1e achieve smaller difference size than PPR (The only difference is deleting `()` from a function call `fn()` in the variant). However, such a difference only indicates that the bug is caused by calling `fn`, which provides far less information than PPR. Moreover, reduced seed and variant given by DD (27,830/27,832 tokens) are several orders larger than those from PPR (89/100 tokens)—the final result still has tens of thousands of tokens, and thus are difficult to be used for debugging.

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

2.1 Practicality

Besides the motivating example, several facts on compiler testing inspire our work and make the concept of PPR practical and general. First, in mutation-based testing, the seed from which a bug-triggering variant is derived is always available. Regardless how drastic the seed is mutated, there are still unchanged code snippets, making differencing algorithms applicable. Second, many changes by mutation are superfluous and irrelevant to the bug, and removing them has no impact on reproducing the bug. Third, the generated variants are usually syntactically valid, and thus can be reduced efficiently and effectively with tree-based program reduction algorithms [28, 37].

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

[2] A miscompilation is a category of compiler bugs, where the buggy compiler silently compiles a well-defined program into a binary that has unexpected semantics.

PPR can minimize the seed, the variant, and the differences between the seed and variant in a single reduction process.

3 PRELIMINARIES AND FORMALIZATION

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

3.1 Classical Program Reduction

Given a program P that has a property, let $\mathbb{P}$ denote the universe of all possible programs that are derivable from P by deleting certain tokens ($P \in \mathbb{P}$ as we can delete zero tokens from P). Let $\mathbb{B} = \{\text{true}, \text{false}\}$, then the property of P can be defined as a predicate $\psi : \mathbb{P} \to \mathbb{B}$; for any program $p \in \mathbb{P}$ (*e.g.*, $p = P$), $\psi(p) = \text{true}$ if p exhibits the property, otherwise $\psi(p) = \text{false}$. As formally defined by Perses [37], the number of tokens in a program p is denoted as $|p|$. The goal of program reduction is to find a program p that satisfies the property and has the fewest tokens in p:

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

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

DDMin. DDMin [48] takes as input a list of elements and a property ψ that the list has, and returns a new list by removing the elements that are irrelevant to ψ from the input list. When applying DDMin to reduce a program with ψ, we can convert the program into a list of lines, tokens, or characters, and let DDMin to remove the ψ-irrelevant elements. However, DDMin does not consider the syntactical constraints of programs during reduction, making it less effective and less efficient than tree-based program reduction.

Tree-Based Program Reduction. Different from DDMin, syntax-guided program reduction [17, 28, 37, 39] converts the program under reduction into a parse tree [1] *w.r.t.* the formal syntax of the program. Such algorithms traverse the tree in a certain order and repeatedly apply DDMin to reduce a list of tree nodes each time. Due to this tree representation of programs, tree-based program reduction algorithms usually run faster than DDMin and yield smaller results.

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

3.2 Differences Between a Pair of Programs

Program differencing algorithms compute the differences between a pair of programs. Based on how programs are represented for differencing, there are mainly two types of differencing algorithms: list-based [9, 30] and tree-based [13, 14, 33].

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

l_1 = [`unsigned` , `int` , `b` , `=` , `1` , `;`]

l_2 = [`int` , `b` , `,` , `c` , `=` , `3` , `;`]

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

Tree-Based Differencing. A tree-based differencing algorithm, denoted as $\Delta_T(t_1, t_2)$, infers the differences between two programs by representing programs as trees. It takes as input two trees t_1 and t_2, and returns a list of operations that converts t_1 to t_2. For example, given two programs in Figure 2, edit operation list [delete $\texttt{Decl}_2$, insert $\texttt{Stmt}_1$, replace `b` with `a` on $\texttt{Expr}_1$] is returned. More details can be found in [13].

Patching. Patching is an important step to convert a program to another based on differences computed by aforementioned algorithms. Given $d = \Delta_L(l_1, l_2)$, patching is the inverse function, converting l_1 to l_2 with the difference d, *i.e.*, $\texttt{Patch}(l_1, d) = l_2$. This algorithm is applicable in two scenarios. First, given d^-, a reduced version of d, $\texttt{Patch}(l_1, d^-)$ returns l_2^-, closer to l_1 than the original l_2. Second, if l_1^- is a subset of l_1, *i.e.*, certain tokens are deleted, $\texttt{Patch}(l_1^-, d)$ returns l_2^- with those tokens removed and differences applied. The patching process for tree-based differences is similar.

3.3 Pairwise Program Reduction

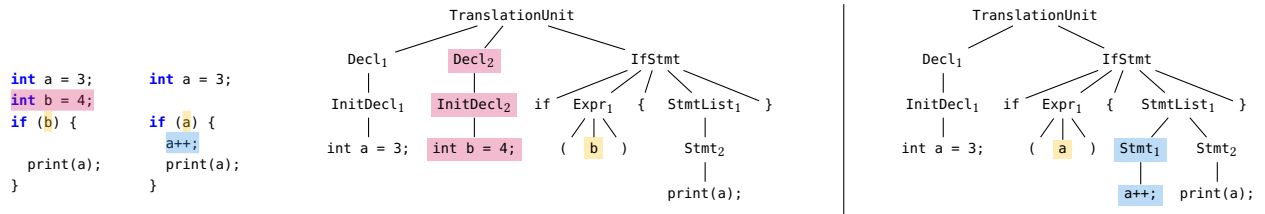
Pairwise program reduction is akin to the classical program reduction but considering a pair of programs. Let P_s and P_v be the original seed and variant programs that exhibits property ψ_s and ψ_v separately, $\mathbb{P}_s$ (and $\mathbb{P}_v$) be the universe of all possible programs derivable from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

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

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

4 APPROACH

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



(a) p_1 (b) p_2 (c) t_1, the parse tree of p_1. (d) t_2, the parse tree of p_2.

Figure 2: Two programs and their tree-based difference. insert, delete and replace from seed and variant are highlighted in blue, red to yellow separately, while the common part is black.

Algorithm 1: PPR $(P_s, P_v, \psi_s, \psi_v)$

Input: P_s: the seed that does not trigger the bug.
Input: P_v: the bug-triggering variant derived from P_s.
Input: $\psi_s : \mathbb{P} \to \mathbb{B}$: the property to be preserved by P_s.
Input: $\psi_v : \mathbb{P} \to \mathbb{B}$: the property to be preserved by P_v.
Output: (P_s^*, P_v^*): the minimized seed and variant with minimal differences.

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

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

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

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

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

To exemplify how the three reducers work, in Figure 3, we use code snippets in GCC-66691 [4] to demonstrate how PPR reduces them step by step. This crash bug in GCC-5.1.0 is discovered with a mutation-based testing technique [36]. As shown in Figure 3a, the original changes include 25 tokens; but after applying PPR, the deletion of `break` statement is proved to be the only critical change highly relevant to the bug, as highlighted in Figure 3d.

Algorithm 2: `MinTdiff`$(P_s, P_v, \psi_s, \psi_v)$

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

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

4.1 Minimizing Tree-Based Differences

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

- (line 5–6) delete a subtree t from P_s: PPR attempts to remove nodes of t from P_s by calling `MinSubtree`(P_s^*, ψ_s, t), so that the unique subtrees from P_s can be minimized. In Figure 3b, `e--;` and `f = g[0][9];` on P_s are removed by this step.
- (line 7–8) insert a subtree t to P_v: Similarly, PPR attempts to remove subtrees of t from P_v to minimize the unique subtree of P_v. In Figure 3b, `while(b == 0);` on P_v is removed by this step.

```
// seed.c
for (a = 0; (a != 22); ++a) {
  int32_t b = 7L;
  int32_t c = 0x2474AFADL;
  int32_t d = (-4L);
  e--;
  f = g[0][9];
  if (b)
    break;

}
```

```
// variant.c
for (a = 0; (a != 22); ++a) {
  int32_t b = 7L;
  int32_t c = 0x2474AFADL;
  int32_t d = (-4L);
  while(b == 0);
  {{
    if (b)
      ;
  }}
}
```

(a) Original programs.

```
for (a = 0; (a != 22); ++a) {
  int32_t b = 7L;
  int32_t c = 0x2474AFADL;
  int32_t d = (-4L);
  e--;
  f = g[0][9];
  if (b)
    break;

}
```

```
for (a = 0; (a != 22); ++a) {
  int32_t b = 7L;
  int32_t c = 0x2474AFADL;
  int32_t d = (-4L);
  while(b == 0);
  {{
    if (b)
      ;
  }}
}
```

(b) After calling `MinTdiff`.

```
for (a = 0; (a != 22); ++a) {
  int32_t b = 7L;
  int32_t c = 0x2474AFADL;
  int32_t d = (-4L);
  e--;
  f = g[0][9];
  if (b)
    break;

}
```

```
for (a = 0; (a != 22); ++a) {
  int32_t b = 7L;
  int32_t c = 0x2474AFADL;
  int32_t d = (-4L);
  while(b == 0);
  {{
    if (b)
      ;
  }}
}
```

(c) After calling `MinLdiff`.

```
for (a = 0; (a != 22); ++a) {
  int32_t b = 7L;
  int32_t c = 0x2474AFADL;
  int32_t d = (-4L);
  e--;
  f = g[0][9];
  if (b)
    break;

}
```

```
for (a = 0; (a != 22); ++a) {
  int32_t b = 7L;
  int32_t c = 0x2474AFADL;
  int32_t d = (-4L);
  while(b == 0);
  {{
    if (b)
      ;
  }}
}
```

(d) After calling `MinCommonality`.

Figure 3: The process of PPR reduces a code snippet of GCC-66691 step by step. Deletion and insertion of tokens are highlighted in red and blue separately. Tokens overlaid by gray are those deleted by PPR.

- (line 9–11) replace a subtree t with s in P_v: PPR attempts to undo the replacement by replacing the subtree s with the subtree t.

Minimizing a Subtree. Given a program P, the property ψ of P, and a *subtree* of P, the function `MinSubtree` removes the descendant nodes of *subtree* that are irrelevant to ψ. `MinSubtree` is similar to existing tree-based reduction techniques such as HDD [28] and Perses [37]. The major difference is that both HDD and Perses start the reduction process from the root node of P, whereas `MinSubtree` starts the reduction process from the root node of the given *subtree*. `MinSubtree` can be implemented by extending HDD or Perses to start reduction from a specified tree node.

4.2 Minimizing List-Based Differences

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

Although both tree-based and list-based difference reduction aim at minimizing difference between P_s and P_v, they complement each other. The former is more efficient by leveraging the formal syntax, the latter adopts DDMin, which is less effective but simpler and more general. In the example, pairs of redundant curly brackets {{ and }} inserted to P_v are removed (Figure 3c). Such changes are neglected by tree differencing algorithms due to the limitation of the used tree differencing algorithms.

Algorithm 3: `MinLdiff`(P_s, P_v, ψ_v)

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

```
1 d ← Δ_L(P_s, P_v)
2 P_v^* ← P_v
  // Diff candidates sorted by size in ascending order.
3 candidates ← all possible deletions by DDMin
4 foreach d ∈ candidates do
5     P_v^* ← Patch(P_s, d)
6     if ψ_v(P_v^*) then
7         break
8 return P_v^*
```

4.3 Minimizing Commonality of P_s and P_v

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

In Figure 3d, the common part is reduced, including bug-irrelevant statement `if(b)` and unused variables `b`, `c`, and `d`.

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

Input: P_s: the seed program, must be parsable.
Input: P_v: the variant program.
Input: $\psi_s : \mathbb{P} \to \mathbb{B}$: the property to be preserved by P_s.
Input: $\psi_v : \mathbb{P} \to \mathbb{B}$: the property to be preserved by P_v.
Output: (P_s^*, P_v^*): the directly minimized seed and variant.

$d \leftarrow \Delta_T(P_s, P_v)$
$P_s^* \leftarrow P_s, P_v^* \leftarrow P_v$
worklist $\leftarrow$ { root node of P_s }
while |*worklist*| > 0 **do**
  $t \leftarrow$ `Pop`(*worklist*)
  `// Seed candidates sorted by size in ascending order.`
  candidates $\leftarrow$ all possible deletions on t
  foreach $(P_s', rest) \in$ *candidates* **do**
    $P_v' \leftarrow$ `Patch`(P_s', d)
    if $\psi_s(P_s') \wedge \psi_v(P_v')$ **then**
      $P_s^* \leftarrow P_s', P_v^* \leftarrow P_v'$
      `Push`(*worklist*, *rest*)
      break
return (P_s^*, P_v^*)

5 EVALUATION

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

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

5.1 Experimental Setup

We implement PPR based on top of Perses [37]. All experiments are conducted on an Ubuntu 20.04 server with Intel Xeon Gold 5217 CPU@3.00GHz and 384GB RAM.

Benchmark-PPR. Since there is no existing benchmark suite for pairwise program reduction, we constructed Benchmark-PPR to answer the above questions. We collected 20 pairs of seed and variant programs from previous mutation-based C compiler testing studies [23, 36]. According to these studies [23, 36], these seeds are generated by CSmith [47] and the variants were derived from the seed programs using a sequence of mutations. Table 1 shows the information of Benchmark-PPR. The columns **Bug ID** shows the bugs that are triggered by the variant programs. The columns **Original** under **Seed Size**, **Variant Size** and **Difference Size** show the original size of seed, variant and their difference.

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

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

5.2 RQ1. Effectiveness of PPR

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

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

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

Columns **DD** of Table 1 show the effectiveness of DD. On average, DD only achieves 7.61% reduction rate on variant programs. For seed and difference, DD cannot produce acceptable results, and reduction rate is not applicable. Our explanation is that DD performs binary search to delete certain consecutive tokens from the variant to derive a subset (seed) that does not trigger the bug, *i.e.*, final seed and differences do not stem from the original ones. Besides, programs with highly nested structure in Benchmark-PPR cannot be effectively handled by DD. Specifically, in 15 out of 20 cases, the difference between the seed and variant programs are reduced to a few tokens, but the size of seed programs are still more than hundreds of thousands of tokens, making it impossible to comprehend and debug. For the other 5 cases, the seed programs are still zero token. In other words, by removing consecutive tokens from the variant, DD cannot always find a syntactically valid seed that does not trigger the bug.

In summary, the usefulness of DD in pairwise program reduction is limited, since "a small difference provides insights only if the whole program is small and comprehensible" [12]. By contrast, PPR

Table 1: The effectiveness of PPR, PPR$_s$, DD, Perses and C-Reduce on Benchmark-PPR, including final size and reduction rate. The cells with "0" under DD of Seed Size indicate that the reductions did not finish in 24h, and thus failed to find out a seed. In the last line, reduction rate R does not apply to Seed and Difference under DD, as they are all derived from Variant, and thus there is no original version to compare to.

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

can effectively reduce the seed and variant programs and minimize the difference between them.

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

5.3 RQ2. Efficiency of PPR

To answer this research question, we measured the time of PPR in reduction and isolation on Benchmark-PPR, as shown in Table 2. To provide developers in-time results, short reduction time is preferred. In all 20 cases, PPR are able to finish the reduction and isolation in 24 hours, ranging from 0.84 hour to 18.38 hours. On average, it takes PPR 4.55 hours to finish the reduction. By contrast, the execution times from DD are particularly polarized. Some cases are finished in a few minutes, while in five cases, DD cannot finish the reduction in 24 hours. This is because removing consecutive tokens blindly from the variant to search for a seed is risky — it may happen to find an appropriate seed and terminate quickly, result in tiny differences but huge seed programs; it may also unluckily fail to find any syntactically valid seed in reasonable time. More importantly, no matter fast or slow, DD always generates unacceptably large programs, making it infeasible to debug with.

We also measured to what extent do PPR brings time overhead than classical program reduction. Table 2 shows that Perses and C-Reduce averagely take 1.29 hours and 2.76 hours in classical program reduction, which is shorter than PPR. This is expected since Perses and C-Reduce only reduce the variant while PPR additionally reduces the seed, and the difference in between. We believe this trade-off is worthwhile since the extra minimum-sized differences provided by PPR can further ease the debugging process. In fact, PPR is even faster than C-Reduce in 10 out of 20 cases.

Table 2: The execution time of PPR, DD, Perses and C-Reduce on Benchmark-PPR.

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

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

5.4 RQ3. Effectiveness of Each Reducer in PPR

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

Table 3: Results when disabling a reducer or changing the order of reducers. For each scenario, we list the average final size and reduction rate. The cells with "-" indicates the reduction did not finish in 24h.

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

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

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

5.5 RQ4. The Impact of the Order of Reducers

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

If `MinCommonality` is invoked ahead of all difference reducers, the average size of final seed and variant are 791.90 and 871.55, which are 2.06x and 2.13x larger than those from the original order shown in Table 3. The difference size is 423.50, which becomes 16x larger than the original and is almost half of the final seed/variant size on average. Further investigation explains such change: if the common code snippets instead of the difference are reduced first, then the remaining difference will take a larger proportion in the program, making tree-differencing algorithms harder to match the common part. Such impact propagates through iterations and finally produces a pair of diverged program, *i.e.*, seed and variant look very different from each other.

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

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

6 DISCUSSION

6.1 Usefulness of PPR in Fuzzer Design

Besides providing insights to compiler developers when debugging, PPR also offers useful feedback to those who are designing fuzzing techniques. GCC-66186 is the example showcased in the paper of Hermes [36], a compiler fuzzing tool, for illustrating the effect of their proposed mutations. The code inserted by Hermes is highlighted in blue.

To identify this bug-triggering part, the authors of Hermes reduced the variant with C-Reduce and then manually searched for patterns from the original mutation set, which involves 17 code blocks, 3,335 tokens in total. Our algorithm, in contrast, automates the procedure, finishes it in 0.85 hour, and presents the same minimized bug-triggering changes shown in Figure 4.

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

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

Figure 4: Final result of GCC-66186 (code snippet). Code highlighted in blue is the final differences from seed to variant.

6.2 Generality on Other Languages

PPR can be parameterized towards any language. To demonstrate PPR's effectiveness on language other than C, we further conducted experiments using two prevalent programming languages, Rust and JavaScript. To build a benchmark suite similar to Benchmark-PPR, we applied random node-level deletion, insertion and replacement on a pool of seed programs to fuzz rustc 1.58.1 and JerryScript-2.4.0 separately, and collected 9 bug-triggering programs and their corresponding seed programs for each language. Then we run PPR and DD on this benchmark suite.

As shown in Table 4, PPR effectively reduces the seed and variant programs by more than 90%. For variants, PPR achieves 91.86% and 94.58% reduction rates on Rust and JavaScript respectively, higher than 82.12% and 69.71% from DD. For differences, PPR reaches 4 tokens on average on both Rust and JavaScript, far lower than 57

Table 4: Results of PPR and DD on Rust and JavaScript.

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

and 24 tokens from DD. Moreover, for all cases in Rust, DD fails to find out a program for seed. Our explanation is that Rust has a strict syntactical or semantical rule for program, so it is not likely to find a valid subset of the variant without the guidance of syntax.

6.3 Threats to Validity

The threat to internal validity mainly lies in the correctness of the implementation of PPR. To mitigate this, authors have carefully reviewed the implementation and tested it on various test cases.

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

7 RELATED WORK

We discuss three lines of related work.

Test Input Minimization and Generalization. Delta Debugging is the foundational technique for systematically minimizing bug-triggering inputs and changes. Hierarchical Delta Debugging [28] further refines this approach by applying DDMin iteratively on different levels of a program's parse tree from top to bottom. Perses [37, 38] enhances reduction efficiency by normalizing grammars and using transformed grammars to guide the reduction process, resulting in more effective reduction compared to HDD. Vulcan [46] further improves the performance by introducing aggressive program transformations. Specific languages also have tailored reduction tools; C-Reduce [35], designed for C/C++ programs, while J-Reduce focuses on Java bytecode [19, 20]. DDSET [16] introduces automated generalization, abstracting concrete programs into a mixture of terminal and abstract symbols to capture failure-inducing patterns. SmartCheck [34] and Extrapolate [3] specialize in generalizing Haskell programs.

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

Failure-Inducing Change Isolation. Delta Debugging sparked a range of related researches [6–8, 12, 50] in the realm of isolating failure-inducing changes. Crisp [8] assists Java developers in locating such changes by enabling iterative selection, application, and undoing of affecting changes to pinpoint failure-inducing ones. AutoFlow [50] employs static and dynamic analysis for automated isolation of failure-inducing changes. Donaldson et al. introduced Spirv-fuzz [12], an effective reduction and deduplication approach for compiler testing, capable of reducing failure-inducing transformation sequences to smaller ones that still trigger failures. Unique in its approach, PPR not only isolates failure-inducing changes but also minimizes commonalities between two programs.

Slicing-Based Fault Localization . Technique sharing similarity with PPR is dual slicing, a slicing-based approach to locate the source of errors in software. By comparing execution traces of programs of different versions, dual slicing techniques [41, 43] can identify the differences and localize the root cause of bugs or regression faults on a program, *e.g.*, Wang *et al.* [41] studies typical programs such as grep and coreutils. However, PPR differs from program slicing *w.r.t.* the study subjects, as it focuses on programs as the input of a compiler/interpreter, instead of programs on their own. Moreover, PPR performs both reduction and isolation on a pair of program inputs, while dual slicing only isolates the root cause of errors in the program. Unlike dual slicing, which requires white-box techniques for execution traces, program reduction methods like PPR are black-box approaches, suitable for bugs within the compilers, rather than in program inputs.

8 CONCLUSION

In this paper, we propose pairwise program reduction (PPR), a novel perspective to complement classical program reduction techniques. Unlike Perses and C-Reduce, PPR takes as input a pair of programs. It reduces not only the seed and variant programs, but also the differences between them. To evaluate PPR, we construct Benchmark-PPR containing programs that trigger bugs in GCC and LLVM. The experiment conducted with Benchmark-PPR shows that final programs from PPR are several orders of magnitude smaller than those from DD, and even comparable to classical program reduction *w.r.t.* size. PPR provides more insightful difference than DD for both compiler developers and fuzzing designers. Finally, evaluation on Rust and JavaScript demonstrates PPR's generality on other languages.

ACKNOWLEDGMENTS

We thank all the anonymous reviewers in ESEC/FSE'23 for their insightful feedback and comments, which significantly improved this paper. This research is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) through the Discovery Grant, a project under Waterloo-Huawei Joint Innovation Lab, and CFI-JELF Project #40736.

REFERENCES

[1] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. 2007. *Compilers: principles, techniques, & tools*. Pearson Education India.

[2] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *NDSS*. https://doi.org/10.14722/ndss.2019.23412

[3] Rudy Braquehais and Colin Runciman. 2017. Extrapolate: generalizing counterexamples of functional test properties. In *Proceedings of the 29th Symposium on the Implementation and Application of Functional Programming Languages*. 1–11. https://doi.org/10.1145/3205368.3205371

[4] GCC Bugzilla. 2015. *Bug 66691*. Retrieved 2022-07-04 from https://gcc.gnu.org/bugzilla/show_bug.cgi?id=66691

[5] LLVM Bugzilla. 2014. *Bug 21467 - clang hangs on valid code at -Os and above on x86_64-linux-gnu*. Retrieved 2022-08-24 from https://bugs.llvm.org/show_bug.cgi?id=21467

[6] Junjie Chen, Jiaqi Han, Peiyi Sun, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. Compiler bug isolation via effective witness test program generation. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 223–234. https://doi.org/10.1145/3338906.3338957

[7] Junjie Chen, Haoyang Ma, and Lingming Zhang. 2020. Enhanced compiler bug isolation via memoized search. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 78–89. https://doi.org/10.1145/3324884.3416570

[8] Ophelia C Chesley, Xiaoxia Ren, and Barbara G Ryder. 2005. Crisp: A debugging tool for Java programs. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, 401–410.

[9] James Coglan. 2017. *The patience diff algorithm*. Retrieved 2022-07-05 from https://blog.jcoglan.com/2017/09/19/the-patience-diff-algorithm

[10] Alastair F Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated testing of graphics shader compilers. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29.

[11] Alastair F Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpiński. 2020. *Spirv-fuzz*. Retrieved 2023-01-30 from https://github.com/google/graphicsfuzz/blob/master/docs/finding-a-vulkan-driver-bug-using-spirv-fuzz.md

[12] 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

[13] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. 313–324. https://doi.org/10.1145/2642937.2642982

[14] Beat Fluri, Michael Wursch, Martin PInzger, and Harald Gall. 2007. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering* 33, 11 (2007), 725–743. https://doi.org/10.1109/TSE.2007.70731

[15] GCC. 2020. *A Guide to Testcase Reduction*. Retrieved 2022-07-04 from https://gcc.gnu.org/wiki/A_guide_to_testcase_reduction

[16] Rahul Gopinath, Alexander Kampmann, Nikolas Havrikov, Ezekiel O Soremekun, and Andreas Zeller. 2020. Abstracting failure-inducing inputs. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 237–248. https://doi.org/10.1145/3395363.3397349

[17] Satia Herfert, Jibesh Patra, and Michael Pradel. 2017. Automatically reducing tree-structured test inputs. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 861–871.

[18] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*. 445–458. https://doi.org/10.5555/2362793.2362831

[19] 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

[20] 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

[21] Gray Kwong, Jesse Ruderman, and Jesse Schwartzentruber. 2022. *Lithium algorithm*. Retrieved 2022-12-24 from https://github.com/MozillaSecurity/lithium

[22] 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

[23] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. *ACM SIGPLAN Notices* 50, 10 (2015), 386–399. https://doi.org/10.1145/2858965.2814319

[24] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F Donaldson. 2015. Many-core compiler fuzzing. *ACM SIGPLAN Notices* 50, 6 (2015), 65–76. https://doi.org/10.1145/2737924.2737986

[25] 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

[26] LLVM. 2022. *How to submit an LLVM bug report*. Retrieved 2022-07-04 from https://llvm.org/docs/HowToSubmitABug.html

[27] David Majnemer. 2014. *InstCombine: Remove infinite loop caused by FoldOpIntoPhi*. Retrieved 2022-07-04 from https://reviews.llvm.org/rG7e2b9882b147bf9c26faa7b04b56884ec444bd64

[28] 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

[29] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. 2013. Compiler testing via a theory of sound optimisations in the C11/C++ 11 memory model. *ACM SIGPLAN Notices* 48, 6 (2013), 187–196. https://doi.org/10.1145/2499370.2491967

[30] Eugene W Myers. 1986. AnO (ND) difference algorithm and its variations. *Algorithmica* 1, 1 (1986), 251–266.

[31] Eriko Nagai, Hironobu Awazu, Nagisa Ishiura, and Naoya Takeda. 2012. Random testing of C compilers targeting arithmetic optimization. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2012)*. 48–53. https://doi.org/10.1145/2499370.2491967

[32] Eriko Nagai, Atsushi Hashimoto, and Nagisa Ishiura. 2014. Reinforcing random testing of arithmetic optimization of C compilers by scaling up size and number of expressions. *IPSJ Transactions on System LSI Design Methodology* 7 (2014), 91–100. https://doi.org/10.2197/ipsjtsldm.7.91

[33] Mateusz Pawlik and Nikolaus Augsten. 2011. RTED: a robust algorithm for the tree edit distance. *arXiv preprint arXiv:1201.0230* (2011). https://doi.org/10.14778/2095686.2095692

[34] Lee Pike. 2014. SmartCheck: automatic and efficient counterexample reduction and generalization. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell*. 53–64. https://doi.org/10.1145/2775050.2633365

[35] 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

[36] 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

[37] 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

[38] Jia Le Tian, Mengxiao Zhang, Zhenyang Xu, Yongqiang Tian, Yiwen Dong, and Chengnian Sun. 2023. Ad Hoc Syntax-Guided Program Reduction. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA, 3–9) *(ESEC/FSE '23)*. ACM, New York, NY, USA, 5. https://doi.org/10.1145/3611643.3613101

[39] Yongqiang Tian, Xueyan Zhang, Yiwen Dong, Zhenyang Xu, Mengxiao Zhang, Yu Jiang, Shing-Chi Cheung, and Chengnian Sun. 2023. On the Caching Schemes to Speed Up Program Reduction. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (January 2023), Article 1, 30 pages.

[40] 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

[41] Haijun Wang, Yun Lin, Zijiang Yang, Jun Sun, Yang Liu, Jinsong Dong, Qinghua Zheng, and Ting Liu. 2019. Explaining regressions via alignment slicing and mending. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2421–2437. https://doi.org/10.1109/TSE.2019.2949568

[42] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 724–735. https://doi.org/10.1109/ICSE.2019.00081

[43] Dasarath Weeratunge, Xiangyu Zhang, William N Sumner, and Suresh Jagannathan. 2010. Analyzing concurrency bugs using dual slicing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*. 253–264. https://doi.org/10.1145/1831708.1831740

[44] Wikipedia. 2022. *Standard Portable Intermediate Representation*. Retrieved 2023-01-25 from https://en.wikipedia.org/wiki/Standard_Portable_Intermediate_Representation

[45] Wikipedia. 2022. *Static single-assignment form*. Retrieved 2022-08-31 from https://en.wikipedia.org/wiki/Static_single-assignment_form

[46] 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

[47] 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.
[48] 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
[49] Mengxiao Zhang, Zhenyang Xu, Yongqiang Tian, Yu Jiang, and Chengnian Sun. 2023. Artifact for "PPR: Pairwise Program Reduction". https://doi.org/10.5281/zenodo.8267114
[50] Sai Zhang, Yu Lin, Zhongxian Gu, and Jianjun Zhao. 2008. Effective identification of failure-inducing changes: a hybrid approach. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. 77–83. https://doi.org/10.1145/1512475.1512492

Received 2023-02-02; accepted 2023-07-27