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

ZHENYANG XU, University of Waterloo, Canada
YONGQIANG TIAN*, University of Waterloo, Canada
MENGXIAO ZHANG, University of Waterloo, Canada
GAOSEN ZHAO, University of Waterloo, Canada
YU JIANG, Tsinghua University, China
CHENGNIAN SUN, University of Waterloo, Canada

Program reduction has demonstrated its usefulness in facilitating debugging language implementations in practice, by minimizing bug-triggering programs. There are two categories of program reducers: language-agnostic program reducers (AGRs) and language-specific program reducers (SPRs). AGRs, such as HDD and Perses, are generally applicable to various languages; SPRs are specifically designed for one language with meticulous thoughts and significant engineering efforts, *e.g.*, C-Reduce for reducing C/C++ programs.

Program reduction is an NP-complete problem: finding the globally minimal program is usually infeasible. Thus all existing program reducers resort to producing 1-minimal results, a special type of local minima. However, 1-minimality can still be large and contain excessive bug-irrelevant program elements. This is especially the case for AGR-produced results because of the *generic* reduction algorithms used in AGRs. An SPR often yields smaller results than AGRs for the language for which the SPR has *customized* reduction algorithms. But SPRs are not language-agnostic, and implementing a new SPR for a different language requires significant engineering efforts.

This paper proposes Vulcan, a language-agnostic framework to further minimize AGRs-produced results by exploiting the formal syntax of the language to perform aggressive program transformations, in hope of creating reduction opportunities for other reduction algorithms to progress or even directly deleting bug-irrelevant elements from the results. Our key insights are two-fold. First, the program transformations in all existing program reducers including SPRs are not diverse enough, which traps these program reducers early in 1-minimality. Second, compared with the original program, the results of AGRs are much smaller, and time-wise it is affordable to perform diverse program transformations that change programs but do not necessarily reduce the sizes of the programs directly. Within the Vulcan framework, we proposed three simple examples of fine-grained program transformations to demonstrate that Vulcan can indeed further push the 1-minimality of AGRs. By performing these program transformations, a 1-minimal program might become a non-1-minimal one that can be further reduced later.

Our extensive evaluations on multilingual benchmarks including C, Rust and SMT-LIBv2 programs strongly demonstrate the effectiveness and generality of Vulcan. Vulcan outperforms the state-of-the-art language-agnostic program reducer Perses in size in all benchmarks: On average, the result of Vulcan contains 33.55%,

*Yongqiang Tian is also affiliated with the Hong Kong University of Science and Technology.

Authors' addresses: Zhenyang Xu, School of Computer Science, University of Waterloo, Canada, zhenyang.xu@uwaterloo.ca; Yongqiang Tian, School of Computer Science, University of Waterloo, Canada, yongqiang.tian@uwaterloo.ca; Mengxiao Zhang, School of Computer Science, University of Waterloo, Canada, m492zhan@uwaterloo.ca; Gaosen Zhao, School of Computer Science, University of Waterloo, Canada, gaosen.zhao@uwaterloo.ca; Yu Jiang, School of Software, Tsinghua University, Beijing, China, jy1989@mail.tsinghua.edu.cn; Chengnian Sun, School of Computer Science, University of Waterloo, Canada, cnsun@uwaterloo.ca.

2475-1421/2023/4-ART97
https://doi.org/10.1145/3586049

21.61%, and 31.34% fewer tokens than that of Perses on C, Rust, and SMT-LIBv2 subjects respectively. Vulcan can produce even smaller results if more reduction time is allocated. Moreover, for the C programs that are reduced by C-Reduce, Vulcan is even able to further minimize them by 10.07%.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**.

Additional Key Words and Phrases: Program Reduction, Automated Debugging, Test Input Minimization

ACM Reference Format:
Zhenyang Xu, Yongqiang Tian, Mengxiao Zhang, Gaosen Zhao, Yu Jiang, and Chengnian Sun. 2023. Pushing the Limit of 1-Minimality of Language-Agnostic Program Reduction. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 97 (April 2023), 29 pages. https://doi.org/10.1145/3586049

1 INTRODUCTION

Given a program P and a property ψ that P exhibits, program reduction produces a minimized program P' that still exhibits ψ. This technique has demonstrated its usefulness in facilitating debugging language implementations (*e.g.*, compilers, interpreters, debuggers) in practice. For example, given a program P that triggers a bug in GCC or LLVM, program reduction reduces P to a smaller one P' that still triggers the same bug. Many compilers and interpreters, such as GCC, Clang and JerryScipt, have explicitly required bug reporters to reduce the bug-triggering programs before submitting bug reports [GCC-Wiki 2020; JerryScript 2022; LLVM 2022; MozillaSecurity 2022].

Program reducers can be classified into two categories: language-agnostic program reducers (AGRs) and language-specific program reducers (SPRs). AGRs are designed to be generally applicable to a wide range of programming languages, such as Delta Debugging (DD) [Zeller and Hildebrandt 2002], Hierarchical Delta Debugging (HDD) [Misherghi and Su 2006], and Perses [Sun et al. 2018]. SPRs are designed to reduce programs in one specific language $\mathcal{L}$ with $\mathcal{L}$-specific algorithmic optimizations, thus usually producing smaller results than AGRs for $\mathcal{L}$. An exemplary program reducer in this category is C-Reduce [Regehr et al. 2012], which implements C/C++ specific transformations with Clang Libtooling [LLVM/Clang 2022].

The goal of program reduction is to find the minimal program that still exhibits the given property ψ. However, obtaining the global minimum is NP-complete [Misherghi and Su 2006; Zeller and Hildebrandt 2002]. To make program reduction practical, all existing program reducers resort to producing 1-minimal results, a type of local minima first proposed by Zeller and Hildebrantdt [Zeller and Hildebrandt 2002]. A reduced program is 1-minimal if and only if deleting any single element from the program makes the program lose the property ψ. The 1-minimality has been demonstrated to be an effective goal. For example, it enables the state-of-the-art AGR Perses to reduce programs from hundreds of thousands of tokens to just hundreds or even tens of tokens quickly [Sun et al. 2018].

Problems with 1-minimality of AGRs. However, the 1-minimal results produced by AGRs can still be large and contain excessive bug-irrelevant program elements, making the reduced programs suboptimal for debugging. Therefore, it remains desirable but also challenging to further reduce results by AGRs beyond 1-minimality. One feasible solution is to design and implement SPRs. For example, C-Reduce tackles this problem by leveraging well-crafted domain-specific program transformations to reduce C/C++ programs [Regehr et al. 2012]. These language-specific transformations, such as source-to-source function inlining, transform programs in different ways from how AGRs do. As a result, C-Reduce can output much smaller results than those from HDD or Perses when reducing C/C++ programs. However, constructing such a language-specific tool requires extensive language-specific knowledge and significant engineering efforts, and the crafted tool usually can work effectively for only one specific programming language. In fact, to the best of our knowledge, only C/C++, Java, and SMT-LIBv2 have C-Reduce, J-Reduce [Kalhauge and Palsberg

2021], and ddSMT [Kremer et al. 2021; Niemetz and Biere 2013] as their language-specific reducers; other programming languages do not have such counterparts.

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

(1) The program transformations in all existing program reducers, especially AGRs, are not diverse enough. To the best knowledge of the authors, all previously proposed AGRs only support deletion-based program transformations, *i.e.*, program transformations that only delete elements from the original program. This limitation traps AGRs early in 1-tree-minimality, and is the main reason that AGRs do not perform as well as SPRs.

(2) Compared to the original bug-triggering program which might contain hundreds of thousands of tokens, the 1-minimal result produced by AGRs are relatively small; thus, performing more deletion-based program transformations with different deletion strategies and even non-deletion-based program transformations *e.g.*, program transformations that do not necessarily reduce the sizes of the programs, becomes time-wise affordable.

Vulcan incorporates an AGR as the main reducer and a series of auxiliary reducers to keep pushing the limit of 1-minimality of the main reducer. Given a program to be reduced, Vulcan first invokes the main reducer to produce a 1-minimal result. Then, one of the auxiliary reducers is invoked to perform program transformations in hope of directly deleting nodes from the 1-minimal result or creating reduction opportunities. Next, the new program produced by the auxiliary reducer is sent back to the main reducer for further reduction until 1-minimality is achieved again. In this way, Vulcan can effectively further minimize the results of AGRs with reasonable computational cost.

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

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

Sub-Tree Replacement. This program transformation is inspired by mutation-based compiler fuzzing, which is a testing technique that randomly generates test programs by mutating existing programs to test the target compiler. Replacing a sub-tree of the program with a new one is a typical mutation operation applied in many existing mutation-based compiler testing work [Aschermann et al. 2019; Wang et al. 2019]. During a fuzzing process, it is common that different generated test programs trigger the same bug. Therefore, we believe by replacing a sub-tree of a 1-minimal program, it is likely to derive different programs inducing the same bug, and such programs might either be smaller or bring potential reduction opportunities.

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

Our extensive evaluation shows Vulcan is effective while being general. First, the experiment conducted with 20 bug-triggering C programs demonstrated that Vulcan can not only effectively further reduce 1-tree-minimal result produced by the state-of-the-art AGR Perses but also further reduce results produced by C-Reduce, the state-of-the-art SPR in reducing C programs. On average, the output of Vulcan contains 33.55% fewer tokens than that of Perses on the C program benchmarks, and Vulcan can further reduce the size of C-Reduce's results on these benchmarks by 10.07% on average. Second, evaluation results on Rust programs and SMT-LIBv2 programs demonstrate the generality of Vulcan on diverse languages. On average, the output of Vulcan contains 21.61% and 31.34% fewer tokens than that of Perses on Rust benchmarks and SMT-LIBv2 benchmarks respectively. Moreover, we conducted ablation studies to comprehend the contribution of each proposed program transformation towards breaking the limit of 1-tree-minimality. The results show all the proposed transformations are helpful to further reduce the program after achieving 1-tree-minimality. At last, by comparing Vulcan with Vulcan$^+$, a variant with more aggressive reduction settings than the default Vulcan, we demonstrated that Vulcan can further trade off execution time for a smaller result. On average, the output of Vulcan$^+$ contains 5.89% fewer tokens than that of Vulcan at the cost of taking 1.81× the execution time of Vulcan to finish.

Contribution. We make the following contributions.

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

2 A MOTIVATING EXAMPLE

In this section, we use a compiler crash bug LLVM-26760 [Bugzilla 2016] as a motivating example to elaborate how Vulcan further reduces a 1-minimal program produced by Perses.

Figure 1a: Result of Perses. Figure 1a shows the program reduced by Perses from the original bug-triggering program for LLVM-26760. Despite being 1-minimal, the program still has 116 tokens remaining for compiler developers to analyze. If this program can be automatically further reduced to a smaller program, it might be easier and faster for the compiler developers to debug and pinpoint the root cause.

Figure 1b: Progress made by replacing an identifier. In Figure 1a, if we replace `l_790` on line 28 with `g_100`, the resulting program can still trigger the same crash bug. More importantly, after this change the assignment statements marked in red (lines 19–21 and 24) can be all eliminated by Perses. Figure 1b shows the further reduced program.

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

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

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

```
typedef signed int8_t;
typedef short int16_t;
typedef int int32_t;
typedef unsigned uint32_t;
int8_t g_100;
int16_t func_33() {
  int8_t l_790;
  int32_t l_919 = 0x24F96B7BL;
  uint32_t l_1052;
  if (l_790)
    for (;;)
      break;
  else for (; l_919; --l_919);
  int32_t l_1081 = 1L;
  int32_t B4o4obl_919 = l_919;
  LABEL_4o4og:;
  l_1052 = l_1052 >> l_1081;
  if (g_100) {
    l_1052 = l_1052 << B4o4obl_919;
    goto LABEL_4o4og;
  }
}
int main() {}
```

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

```
typedef signed int8_t;
typedef int int32_t;
typedef unsigned uint32_t;
int8_t g_100;
short func_33() {
  int8_t l_790;
  int32_t l_919 = 0x24F96B7BL;
  uint32_t l_1052;
  if (l_790) for (;;) break;
  else for (; l_919; --l_919);
  int32_t l_1081 = 1L;
  int32_t B4o4obl_919 = l_919;
  LABEL_4o4og:;
  l_1052 = l_1052 >> l_1081;
  if (g_100) {
    l_1052 = l_1052 << B4o4obl_919;
    goto LABEL_4o4og;
  }
}
int main() {}
```

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

```
typedef signed int8_t;
typedef int int32_t;
typedef unsigned uint32_t;
int8_t g_100;
short func_33() {
  int8_t l_790;
  int32_t l_919 = 0x24F96B7BL;
  uint32_t l_1052;
  if (l_790) for (; l_919; --l_919);
  int32_t l_1081 = 1L;
  int32_t B4o4obl_919 = l_919;
  LABEL_4o4og:;
  l_1052 = l_1052 >> l_1081;
  if (g_100) {
    l_1052 = l_1052 << B4o4obl_919;
    goto LABEL_4o4og;
  }
}
int main() {}
```

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

```
typedef unsigned uint32_t;
short l_1052() {
  uint32_t main = 0x24F96B7BL;
  uint32_t l_1052;
  for (; l_1052; --main)
  ;
  uint32_t uint32_t = 1L;
  main:;
  l_1052 = l_1052 >> uint32_t;
  l_1052 = l_1052 << main;
  goto main;
}
int main() {}
```

(e) Result produced by Vulcan.

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

However, with this replacement, these statements become redundant as `l_790` is not used, and thus are eliminated by Perses.

Figure 1c: Progress made by replacing a sub-tree. In the program in Figure 1b, if the type specifier `int16_t` marked in orange (a sub-tree in the view of tree representation) is replaced with a different one (*e.g.*, `short`), the `typedef` statement on line 2 can be eliminated without losing the property as the only usage of the defined alias is removed. By applying Perses on the program after the replacement, a smaller program shown as Figure 1c is obtained. Without this replacement, Perses cannot further reduce the program in Figure 1b, because Perses performs only deletion-based program transformation.

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

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

3 BACKGROUND

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

3.1 Program Reduction

Given a program P and a property ψ that P exhibits, the task of program reduction is to explore the search space $\mathbb{P}$ in order to find a minimal P' which still exhibits ψ, where the search space $\mathbb{P}$ is defined by concrete program reduction algorithm [Sun et al. 2018].[1] Mathematically, ψ is a function that maps the program search space $\mathbb{P}$ to $\mathbb{B} = \{\text{true}, \text{false}\}$, such that $\psi(p) = \texttt{true}$ if p exhibits the property, otherwise $\psi(p) = \texttt{false}$. The goal of program reduction is to find the minimal program p_{min} defined as follows:

$$p_{min} \in \{p | \psi(p) \land p \in \mathbb{P} \land \forall p' \in \mathbb{P}. |p'| < |p| \Rightarrow \neg\psi(p')\} \tag{1}$$

where $|p|$ denotes the size of p, which is usually measured according to the number of tokens in p. Please note that there could be multiple p_{min} satisfying the above requirements.

3.1.1 Delta Debugging. Delta Debugging (DD) is the first work that formulated a generic approach for test-case reduction [Zeller and Hildebrandt 2002]. It proposed an algorithm named `ddmin` to minimize a failure-inducing test case. `ddmin` treats the test case as a list of elements and keeps testing different subsets of this list to find a smaller failure-inducing test case. Specifically, `ddmin` first splits the list into n partitions. Then, it traverses each partition to test whether there is a partition that can solely trigger the failure. If so, it removes all the other partitions and starts over from the obtained smaller test case. Otherwise, it traverses all the partitions again to test whether there is a partition whose complement can trigger the failure. If so, it removes the partition to obtain a smaller test case and restarts. If the test case cannot be reduced during the above

[1] $\mathbb{P}$ is the universe of candidate programs that a program reducer can derive from P, instead of the universe of all programs.

process, ddmin further splits the list into $2n$ partitions and repeats the process. ddmin can minimize bug-triggering programs by considering them as a list of tokens or lines. However, since ddmin does not leverage the structures of programs, its efficiency and effectiveness in program reduction are suboptimal [Misherghi and Su 2006; Sun et al. 2018].

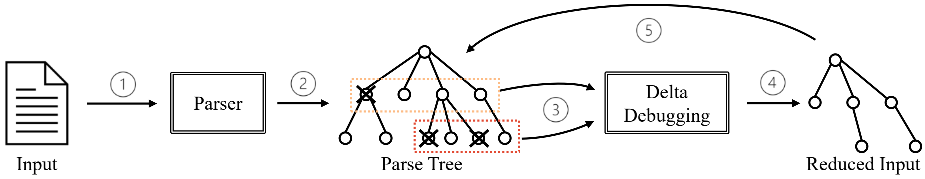


Fig. 2. An example workflow of tree-based program reduction.

3.1.2 Tree-Based Program Reduction Approach. Hierarchical Delta Debugging (HDD) is the first tree-based program reduction proposed by Misherghi and Su [Misherghi and Su 2006]. HDD aims to speed up ddmin, especially the performance when handling complex and structured inputs, by leveraging the tree structure information of the input. Figure 2 illustrates the workflow of HDD. First, the input program is parsed to a parse tree by a language parser (step ① and ②). Next, the parse tree is reduced in a top-down manner. Specifically, HDD performs ddmin to the parse tree level by level to eliminate bug-irrelevant elements from coarsest to finest (step ③). After applying the ddmin algorithm to each level of the parse tree, a reduced input is obtained (step ④). It should be noted that the reduced input may be able to further reduced by the same tree-based reduction algorithm since the deletion of some nodes may enable other nodes to be deleted. Therefore, in practice, to obtain a small program as much as possible, a tree-based program reduction algorithm is usually run in *fixpoint mode*, in which the algorithm is iteratively applied on the reduced input until the size can no longer be smaller (step ⑤).

Perses [Sun et al. 2018] further leverages language syntactic rules to facilitate the reduction. Specifically, it utilizes context-free grammar to ensure the derived program variants are always syntactically valid and avoid futile efforts on syntactically invalid variants. Since it is the state-of-the-art reducer in terms of both effectiveness and efficiency, we use it as our baseline.

3.2 Relaxed Goal in Program Reduction

Finding the minimal program p_{min} defined in §3.1 is NP-complete [Misherghi and Su 2006; Zeller and Hildebrandt 2002]. In practice, to complete the reduction in a reasonable time, this goal is usually relaxed by existing reducers to local minima, *i.e.*, 1-minimality or 1-tree-minimality [Heo et al. 2018; Misherghi and Su 2006; Sun et al. 2018; Wang et al. 2021; Zeller and Hildebrandt 2002].

1-Minimality. 1-minimality is proposed by DD [Zeller and Hildebrandt 2002]. Specifically, a 1-minimal program $p_{1\text{-}min}$ is a local minimal solution: (1) $p_{1\text{-}min}$ passes the property test ψ and (2) any program variant derived from $p_{1\text{-}min}$ by deleting a single element (*e.g.*, a token) cannot pass the property test ψ. It can be formally defined as follows:

$$p_{1\text{-}min} \in \{p|\psi(p) \land p \in \mathbb{P} \land \forall p' \in \mathbb{P}.|p| - |p'| = 1 \Rightarrow \neg\psi(p')\} \tag{2}$$

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

$$p_{\textit{1-tree-min}} \in \{p|\psi(p) \land p \in \mathbb{P} \land \forall p'.\textit{simplify}(T(p), T(p')) \Rightarrow \neg\psi(p')\} \tag{3}$$

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

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

4 APPROACH

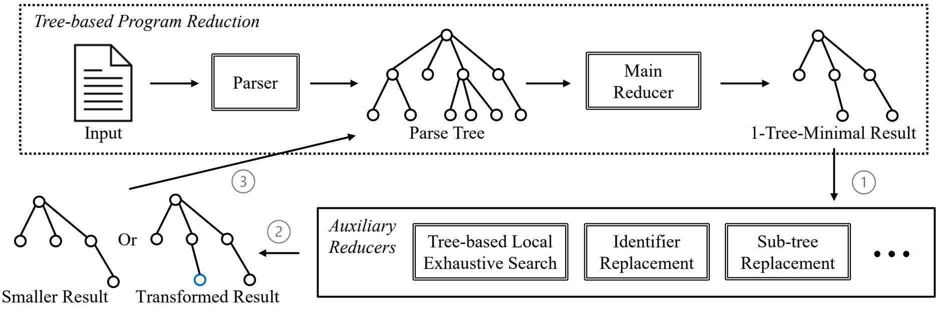


Fig. 3. The workflow of Vulcan.

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

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

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

Initialization. First, Vulcan initializes a series of variables. minimum is the current program satisfying ψ. It is initialized with the program to be reduced P (line 1). prev_by_main is the program from which minimum is derived. It is initialized with null (line 2). index is the index of the current auxiliary reducer being used, and it is initialized to 0 (line 3). attempts records the number of

Algorithm 1: Workflow of Vulcan

```
   Input  : $P$, the program to be reduced.
   Input  : $\psi$: $\mathbb{P} \to \mathbb{B}$, the property checking function
   Output : A minimized program $p \in \mathbb{P}$ $s.t.$ $\psi(p)$
   Data: main_reducer: the main program reducer
   Data: aux_reducers: the list of the auxiliary reducers
1  minimum ← $P$
2  prev_by_main ← null
3  index ← 0
4  attempts ← 0
5  while true do
       // run the main reducer to produce a 1-minimal result
6      prev_by_main ← minimum
7      minimum ← main_reducer.Reduce(minimum, $\psi$)
8      if |minimum| < |prev_by_main| then
9          attempts ← 0
10         prev_by_main ← minimum
       // Start running auxiliary reducers
11     while minimum == prev_by_main ∧ index < |aux_reducers| do
12         minimum ← aux_reducers[index].Reduce(minimum, $\psi$)
13         attempts ← attempts + 1
14         if minimum == prev_by_main then
15             index ← index + 1
16             attempts ← 0
17     if |minimum| < |prev_by_main| then
           // If progress was directly made by the auxiliary reducer
18         attempts ← 0
19     else if attempts > $\sigma_{limit}$ then
20         index ← index + 1
21         attempts ← 0
22     if index >= |aux_reducers| then return minimum
```

attempts of the current auxiliary reducer. Whenever Vulcan switches to the next auxiliary reducer or the current auxiliary reducer helps advance the reduction progress, attempts is reset. It is initialized to 0 (line 4).

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

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

- If minimum is smaller than prev_by_main (*i.e.*, progress has been directly made by the auxiliary reducer) (line 17), attempts is reset (line 18), and Vulcan invokes the main reducer again to reduce this smaller program (line 6-10).
- If minimum is equal in size to prev_by_main (*i.e.*, progress can potentially be made by the main reducer from this new minimum), Vulcan first checks the value of attempts. If attempts is larger than a pre-set limit σ_{limit}, Vulcan switches to the next auxiliary reducer (line 20), resets the attempts to 0 (line 21), and terminates if all the auxiliary reducers have been used (line 22). Otherwise, Vulcan invokes the main reducer again to reduce the new program produced by the auxiliary reducer (line 6-10).

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

4.1 Identifier Replacement

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

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

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

Algorithm 2: Identifier Replacement

```
Input  : $P$, the program to be transformed.
Input  : $\psi$: $\mathbb{P} \to \mathbb{B}$, the property checking function
Output : A transformed program $p$, $s.t.$ $\psi(p)$
$t \leftarrow$ ParseTree ($P$)
id_clusters $\leftarrow$ find all the identifier nodes in $t$ and cluster them by their name.
foreach cluster $\in$ id_clusters do
    foreach another_cluster $\in$ id_clusters \ { cluster } do
        id_name $\leftarrow$ the identifier name of another_cluster
        $t' \leftarrow$ a new parse tree by replacing the identifiers in cluster except the first one with id_name
        $p \leftarrow$ program derived from $t'$
        if $\psi(p)$ then return $p$
return $P$
```

reduction process, most time is spent on the executions of property tests [Sun et al. 2018]. Given a program, the number of property tests directly performed by *Identifier Replacement* is at most $n_c(n_c - 1)$ where n_c is the number of clusters (sets of identifiers that have the same name). If it succeeds in finding a program passing the property test, the main reducer needs to be invoked subsequently. The property tests required by the tree-based reduction algorithm may vary based on which specific algorithm is used in the framework. In practice, the computational complexity of HDD is $O(n^2)$ at worst and $O(n)$ in typical cases approximately where n represents the size of the parse tree [Misherghi and Su 2006]. Perses has the same worst computational complexity though it requires fewer property tests on average than HDD [Sun et al. 2018]. If the main reducer fails to reduce the size, *Identifier Replacement* will produce another program and send it to the main reducer again. This process is continued until the 1-minimal result is successfully reduced or the pre-set limit of failed attempts is reached or *Identifier Replacement* can no longer find programs that can pass the property test. Consequently, let t be the number of attempts, which is limited to a small value by a pre-set parameter, the worst computational complexity of *Identifier Replacement* can be represented as follow:

$$O(t(n_c(n_c - 1) + n^2)) = O(n_c^2 + n^2) = O(n^2) \tag{4}$$

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

4.2 Sub-Tree Replacement

This transformation is inspired by mutation-based compiler fuzzing [Aschermann et al. 2019; Donaldson et al. 2017; Le et al. 2014, 2015; Sun et al. 2016], a testing technique that randomly generates test programs by mutating existing programs and make the target compiler compile these generated programs in the hope of some unexpected behavior can be triggered, thus detecting bugs. Replacing a sub-tree of the program's parse tree with a new one is a typical mutation operation applied in many existing work [Aschermann et al. 2019; Wang et al. 2019] During a fuzzing process, it is common that different generated test programs trigger the same bug. Therefore, we believe that by replacing a sub-tree of a bug-triggering program, it is likely that a different program that triggers the same bug can be obtained, and this different program might either be smaller or bring potential reduction opportunities. To sum up, *Sub-Tree Replacement* is designed to find different programs that trigger the same bug like how it happens in the compiler fuzzing process.

Before describing the transformation, we first introduce some concepts and knowledge about parse trees and formal grammar in this paragraph. Each non-leaf node of a parse tree corresponds to a non-terminal symbol in the grammar. Sub-trees whose roots correspond to the same non-terminal symbol are syntactically compatible structures. In a language grammar, some non-terminal symbols have different production rules, which means they can have different forms. For example, a non-terminal symbol *typeSpecifier* can be `int`, `char`, or any other type, and a symbol *statement* may refer to a *if statement* or *while statement*. We refer to these different forms of a non-terminal symbol as its *alternatives*. Replacing a sub-tree with a different alternative of the non-terminal symbol corresponded by the root does not break syntactical validity. Given a grammar and a specific non-terminal, a sub-tree can be generated by keeping selecting and applying production rules from the given non-terminal symbol until only terminal symbols are left [Aschermann et al. 2019; Kreutzer et al. 2020].

Algorithm 3: Sub-Tree Replacement

```
   Input  : $P$, the program to be transformed.
   Input  : $\psi$: $\mathbb{P} \to \mathbb{B}$, the property checking function
   Output : A transformed program $p$, $s.t.$ $\psi(p)$)
1  $t \leftarrow$ ParseTree ($P$)
2  alternative_nodes $\leftarrow$ All the nodes that correspond to non-terminal symbols with multiple production
     rules in $t$
3  foreach node $\in$ alternative_nodes do
4      $t_{sub} \leftarrow$ generate a sub-tree rooted in the same non-terminal as node
5      $t' \leftarrow$ replace sub-tree rooted in node with $t_{sub}$ in $t$
6      $p \leftarrow$ program derived from $t'$
7      if $\psi(p)$ then  return $p$
8  return $P$
```

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

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

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

$$O(t(n \times n_a + n^2)) = O(n^2) \tag{5}$$

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

4.3 Tree-Based Local Exhaustive Enumeration

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

Algorithm 4: Tree-Based Local Exhaustive Enumeration

```
Input  : $P$, the program to be transformed.
Input  : $\psi$: $\mathbb{P} \rightarrow \mathbb{B}$, the property checking function
Output : A reduced program $p, s.t.\ \psi(p)$
$t \leftarrow$ ParseTree ($P$)
node_sequence $\leftarrow$ [root node of $t$ ]
while length of node_sequence < $\sigma_{wsize}$ do
    node_sequence $\leftarrow$ nodes in the next level of $t$
while true do
    window_position $\leftarrow$ 0
    while window_position +$\sigma_{wsize}$ < length of node_sequence do
        node_set $\leftarrow$ node_sequence[window_position: window_position +$\sigma_{wsize}$]
        proper_subsets $\leftarrow$ all true subsets of node_set
        filtered_proper_subsets $\leftarrow$ $\{s \mid s \in$ proper_subsets $\wedge |$node_set $\setminus s| \geq 2\}$
        foreach $s \in$ filtered_proper_subsets do
            nodes_to_remove $\leftarrow$ node_set $\setminus s$
            $t' \leftarrow$ a new tree obtained by removing nodes in nodes_to_remove from $t$
            $p \leftarrow$ program derived from $t'$
            if $\psi(p)$ then
                reduced $\leftarrow$ true
                $t \leftarrow t'$
                remove the removed nodes from node_sequence
            else window_position $\leftarrow$ window_position + 1
    if all nodes in node_sequence are leaf nodes then break
    node_sequence $\leftarrow$ nodes in the next level of $t$
return $p$
```

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

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

Figure 4 illustrates how sliding window moves in *Tree-Based Local Exhaustive Enumeration* with a concrete example. At the beginning, there is only one node which is the root of the parse tree in the node sequence. When the number of nodes is fewer than the length of the sliding window, the algorithm updates the node sequence with nodes in the next level (*i.e.*, replace all the non-leaf nodes in the node sequence with their children). In this case, the root has four children, and the length of the sliding window is three, so we update the node sequence to contain the four children of the root and make the initial sliding window include the first three nodes in the node sequence. When the sliding window has not reached the end of the node sequence, it moves forward by 1 node each time (step ① and ③ in Figure 4). Otherwise, the node sequence gets updated by replacing all non-leaf nodes with their children again (step ② in Figure 4) and the sliding window is again moved to the beginning of the node sequence.

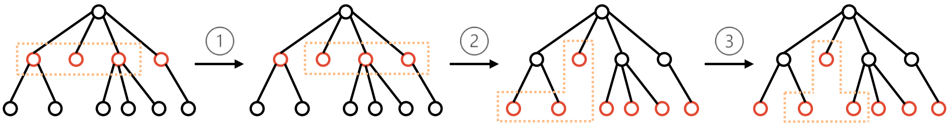


Fig. 4. An illustrative example that shows how the sliding window (marked in orange box, length = 3) moves during the process of tree-based local exhaustive search. If the window has not reached the end of the current node sequence (marked with red), it moves forward by one node (step ① and ③). Otherwise, the node sequence is updated by replacing all non-leaf nodes with their children (step ②).

Computational Complexity Analysis. Different from the *Identifier Replacement* and *Sub-Tree Replacement*, *Tree-Based Local Exhaustive Enumeration* is designed to directly reduce elements from the program. The computational complexity of this auxiliary reducer depends on the size of the tree and the window size. Suppose the parse tree to be reduced contains n nodes with a constant branching factor b, and the depth of the tree is h. Also, let the window size equal to w.

Then, in the worst case, the number of property tests required by this auxiliary reducer equals to $\sum_{i=1}^{h} max(b^i - w, 0) * (2^w - w)$, which can be considered as $O(n)$ at worst when w is limited to a small value. For the previous two auxiliary reducers, it might take several rounds of alternate invocation of the auxiliary reducer and the main reducer to eventually make progress on further reduction. However, for *Tree-Based Local Exhaustive Enumeration*, multiple executions are meaningless, and Vulcan will immediately switch to the next auxiliary reducer or terminate once the invocation of *Tree-Based Local Exhaustive Enumeration* does not make progress.

4.4 Order of Auxiliary Reducers

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

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

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

4.5 Parameters and Fixpoint Mode

There are two parameters that can be adjusted to control the behavior of the auxiliary reducers. For auxiliary reducers that perform non-deletion-based program transformations, the parameter σ_{limit} limits the maximum number of attempts an auxiliary reducer can make before successfully further reducing the 1-minimal result. If the auxiliary reducer continuously produces σ_{limit} programs and none of them is either smaller or able to be further reduced by the main reducer, Vulcan will switch to the next auxiliary reducer. For the auxiliary reducer perform *Tree-Based Local Exhaustive Enumeration*, the parameter σ_{wsize} decides the size of the sliding window. Both parameters can affect the performance of Vulcan. Within a certain range, when σ_{limit} increases, the result produced by Vulcan will tend to be smaller since auxiliary reducers which perform non-deletion-based program transformations will try more times to search for reduction opportunities. However, the execution time of Vulcan will also be increased due to the same reason. Similarly, when σ_{wsize} increases, the number of programs that *Tree-Based Local Exhaustive Enumeration* generates will increase, thus can potentially produce smaller results at the cost of longer execution time.

In program reduction, it is common to perform the reduction algorithm multiple times to achieve smaller results. This is possible because the deletion of some elements may enable other elements to be deleted. AGRs including HDD and Perses run in a so-called fixpoint mode to promise 1-minimality [Misherghi and Su 2006; Sun et al. 2018]. In this mode, the reduction algorithm is kept performing until the program can no longer be reduced (*i.e.*, reaching a fixed point). The main reducer of Vulcan should always run in fixpoint mode to directly produce a 1-minimal result. Besides, Vulcan can also run its auxiliary reducers in fixpoint mode. By default, Vulcan terminates after all the auxiliary reducers have been used. However, it is possible that the first auxiliary reducer can continue to help further reduction after the program is reduced with the help of the subsequent auxiliary reducers. Therefore, by using auxiliary reducers iteratively rather than using them only once, Vulcan can trade off the execution time for a potentially smaller result.

Table 1. Number of queries required by performing Tree-Based Local Exhaustive Enumeration with different values of σ_{wsize}. $Q_{pos}(\sigma_{wsize})$ is the number of queries required by each position of the sliding window given the σ_{wsize}. $Q_{level}(\sigma_{wsize})$ is the number of queries required by the lowest level of a parse tree that has s leaf nodes given the σ_{wsize}.

σ_{wsize}	$Q_{pos}(\sigma_{\text{wsize}})$	$Q_{level}(\sigma_{\text{wsize}})$			
		$s = 100$	$s = 200$	$s = 300$	$s = 400$
2	1	99	199	299	399
3	4	392	792	1192	1592
4	11	1067	2167	3267	4367
5	26	2496	5096	7696	10296
6	57	5415	11115	16815	22515
7	120	11280	23280	35280	47280
8	247	22974	47671	72371	97071

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

For σ_{wsize}, we choose its default value by roughly analyzing the number of queries required by performing *Tree-Based Local Exhaustive Enumeration* with different values of σ_{wsize}. Specifically, for each value of σ_{wsize}, we calculated the number of queries required by each position of the sliding window and the lowest levels of the parse trees derived from programs with different sizes. As shown in Table 1, the number of queries required increases exponentially as σ_{wsize} grows, which makes the affordable choices of this parameter very limited. We think setting σ_{wsize} to 2 or 3 is too conservative. 4 or 5 seems to be the balanced choice for σ_{wsize}, and when the value of σ_{wsize} is larger than 6, the runtime overhead becomes unaffordable. Based on the above analysis, we finally set σ_{wsize} to 4 as the default value mainly for efficiency reasons.

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

5 EVALUATION

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

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

To answer this question, we measured the effectiveness of Vulcan on the benchmarks used in Perses [Sun et al. 2018], which contains 20 real-world complex bug-triggering C programs generated by fuzzing techniques, including CSmith [Yang et al. 2011] and EMI [Le et al. 2014], and compared it with two state-of-the-art tools, Perses and C-Reduce.

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

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

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

To demonstrate the generality of Vulcan, we measured the performance of Vulcan in two languages other than C/C++, *i.e.*, Rust and SMT-LIBv2. We collected 13 Rust programs by ourselves and 93 SMT-LIBv2 programs from ddSMT2.0 [Kremer et al. 2021]. We compared the performance of Vulcan with Perses, the state-of-the-art AGR.

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

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

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

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

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

5.1 RQ1: Further Reduction on 1-minimal Results

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

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

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

Unsurprisingly, Vulcan does not outperform C-Reduce in terms of both size and execution time. On average, the results of Vulcan contain 80.54% more tokens than those of C-Reduce. This is expected since C-Reduce is specifically designed for reducing C/C++ programs and it has been constantly tuned and improved for a long time since it was first designed [Regehr et al. 2022]. It should be noted that Vulcan is designed to help debug language processors where there is no such a language-specific tool to reduce programs in the corresponding programming language. Nevertheless, there are still two subjects, *gcc-71626* and *gcc-70586*, in which Vulcan produces smaller results than C-Reduce.

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

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

Table 2. Evaluation results of Perses, C-Reduce, and Vulcan on C programs. The size of a program is measure by the number of tokens it contains. O(#) is the size of the **O**riginal bug-triggering program. R(#) is the size of the **R**educed program. Q(#) is the number of required property test **Q**ueries. T(s) is the execution **T**ime in seconds. C(%) is the percentage **C**hanges in size of Vulcan's results *w.r.t.* Perses and C-Reduce.

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

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

Table 3. Evaluation results of Vulcan and Perses on C-Reduce reduced C programs. C(%) is the percentage **C**hange in size *w.r.t.* the result of C-Reduce.

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

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

The results are shown in Table 3. Overall, Vulcan further reduces C-Reduce's results in 18 out of 20 subjects. On average, 10.07% of the tokens are further reduced by Vulcan within around 22 minutes. Moreover, in 7 subjects, the percentage decrease in size *w.r.t.* C-Reduce is over 10%, In the best case *gcc-60116*, 29.76% tokens are further reduced by Vulcan.

To ensure that the potential improvement brought by Vulcan is contributed by our entire framework instead of the main reducer Perses, we also reduced C-Reduce's results with Perses. Table 3 shows the results. We found that Perses can only reduce 5 out of 20 subjects. In 4 out of these 5 subjects, Vulcan's results are much smaller than Perses's results, and they have the same size in the remaining one case, *i.e. clang-23353*. Over these 5 subjects, the average percentage change in size of Perses's result *w.r.t.* C-Reduce is only 5.36%, which is much smaller than Vulcan. Therefore, we believe that the potential improvement brought by Vulcan is contributed by our entire framework instead of the main reducer Perses.

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

5.3 RQ3: Generality of Vulcan

To demonstrate the generality of Vulcan in different programming languages, we evaluated Vulcan on Rust programs and SMT-LIBv2 programs. In both experiments, we use Perses as our baseline, as it is the state-of-the-art AGR. For Rust programs, we also add C-Reduce as a baseline. Although C-Reduce is designed for C/C++ programs, it can also reduce the program in other languages using its language-agnostic line-level and token-level reduction algorithms. According to the authors of C-Reduce, C-Reduce happens to perform well in reducing Rust programs [Regehr et al. 2022]. We also tried to use C-Reduce to reduce SMT-LIBv2 programs. However, our preliminary evaluation shows that C-Reduce cannot effectively reduce most of these programs within a reasonable duration. While both Perses and Vulcan can reduce almost all the bug-triggering programs that contain hundreds of thousands of tokens to a program only containing hundreds of tokens in 1 hour, C-Reduce can only remove hundreds or even tens of tokens after 2 hours in such subjects. Therefore, C-Reduce is not included in the experiment with SMT-LIBv2 benchmarks.

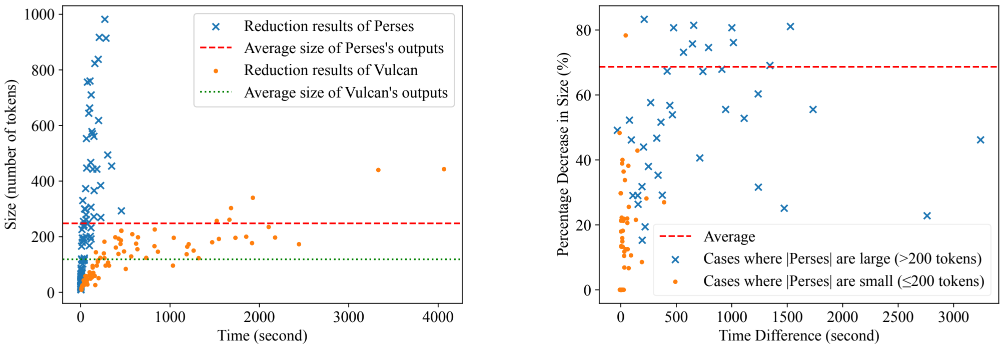
Table 4. Evaluation results of Perses, C-Reduce, and Vulcan on Rust programs. C(%) is the percentage **C**hanges in size of Vulcan's results *w.r.t.* Perses and C-Reduce.

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

5.3.1 *Evaluation on Rust Programs.* To evaluate the performance of Vulcan on Rust programs, we built a set of benchmarks consisting of 13 real-world bug-triggering Rust programs from the issue tracking system of Rust [Rust 2022]. We then used C-Reduce, Perses, and Vulcan to reduce the original programs. The results are shown in Table 4.

Overall, Vulcan outperforms both Perses and C-Reduce in terms of the size of the reduced programs. On average, the result output by Vulcan contains 21.61% and 14.63% fewer tokens than those output by Perses and C-Reduce, respectively. This significant improvement is particularly evident in the case of the bug-triggering program *rust-78005*, where the result produced by Vulcan contains 88.24% and 84.00% fewer tokens than those produced by Perses and C-Reduce, respectively. Additionally, it is important to note that the most valuable feature of Vulcan is its ability to further reduce programs that cannot be sufficiently reduced by previous AGRs. If we only consider cases where the programs still contain more than 50 tokens after being reduced by Perses, the percentage changes in size of Vulcan's results *w.r.t.* Perses and C-Reduce are 27.95% and 19.84%, respectively. In terms of execution time, Vulcan can finish reduction in 2 hours for all subjects, and in 10 out of 13 subjects, the reduction is finished in less than 5 minutes.

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



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

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

Fig. 5. Evaluation results of Perses and Vulcan on SMT-LIBv2 benchmarks.

5.3.2 Evaluation on SMT-LIBv2 Programs. The programs we used for this experiment are from ddSMT2.0 [Kremer et al. 2021]. We filtered out the programs that cannot be parsed because they cannot be handled by the baseline Perses, a tree-based program reduction tool. In total, 93 programs are collected for our evaluation and their numbers of tokens range from 15 to 120,757 (average 6,917).

Figure 5a shows the experiment results. Each cross or dot in the figure represents a reduction result, where its x-coordinate is the execution time of the reduction, and the y-coordinate is the size of the reduced program. To better visualize the difference between Perses's and Vulcan's results for each subject, we calculated the percentage decrease in size that Vulcan achieves *w.r.t.* Perses for each subject and the extra time required by Vulcan in reduction. The comparison results are shown in Figure 5b. On average, the percentage decrease in size of Vulcan *w.r.t.* Perses is 31.34%. In 75 out of 93 subjects, the output of Vulcan is smaller than that of Perses, and in 23 subjects, the size of Vulcan's output is smaller than half of the size of Perses's output. In terms of execution time, in 91 out of 93 subjects, the extra time required by Vulcan is less than half an hour, and in the other two subjects, Vulcan can finish within an extra hour.

As mentioned previously, Vulcan is most useful when previous AGRs cannot sufficiently reduce the program. To investigate how Vulcan performs in this intended application scenario, we excluded the cases where the results output by Perses contain fewer than 200 tokens, and evaluated Vulcan with the remaining 40 cases. In these 40 cases, the output of Vulcan is consistently smaller than that of Perses, and in 22 cases, the size of Vulcan's output is smaller than half of the size of Perses's output. Moreover, on average, in the intended application scenario of Vulcan, the percentage decrease in size is 52.01%, which further demonstrates the effectiveness of Vulcan.

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

5.4 RQ4: Effectiveness of Each Auxiliary Reducer

Table 5. Results of Vulcan$_{EE}$, Vulcan$_{IR}$ and Vulcan$_{SR}$ reducing the 1-tree-minimal program output by Perses. E(#) is the number of **E**liminated tokens in reduction.

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

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

Table 5 shows the results of all the three variants reducing the C programs that have been reduced by Perses. In general, all three variants can reduce the size of Perses's results in some subjects of the C program benchmarks. Specifically, Vulcan$_{EE}$ has the shortest execution time and it succeeds in reducing tokens in 17 out of 20 subjects. Vulcan$_{IR}$ succeeds in reducing tokens in all subjects, and it reduces around 26% of the tokens in Perses's results on average. Vulcan$_{SR}$ reduces tokens in 8 out of 20 subjects, which is not as good as the other two variants but still effective. Interestingly, in four subjects, Vulcan$_{IR}$ eliminates more tokens than Vulcan. We believe this is because Vulcan utilizes *Tree-Based Local Exhaustive Enumeration* before *Identifier Replacement*, and the effect of *Tree-Based Local Exhaustive Enumeration* kills some better potential reduction opportunities which originally can be caught by *Identifier Replacement*.

We also measured the reduction speed of each variant, *i.e.*, the number of eliminated tokens per second. On average, Vulcan$_{EE}$ can reduce 0.029 tokens per second, which is the fastest one among all the three variants. The second fastest one is Vulcan$_{IR}$, which can reduce 0.014 tokens per second. Vulcan$_{SR}$ is the slowest, which only reduces 0.005 tokens per second. This result conforms to our analytical result introduced in §4.4.

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

5.5 RQ5: Performance with a More Aggressive Reduction Setting

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

We evaluated Vulcan$^+$ on the same C programs in RQ1. The results are shown in Table 6. Overall, the reduced program of Vulcan$^+$ contains 5.89% fewer tokens than that of Vulcan at the cost of requiring around 1.8× execution time. In 14 out of 20 subjects, Vulcan$^+$ succeeds in producing a smaller result than Perses. Notably, in 7 subjects, the reduced program of Vulcan$^+$ is 10% smaller than that of Vulcan, and in the subject *clang-23309*, this percentage reaches 27.06%.

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

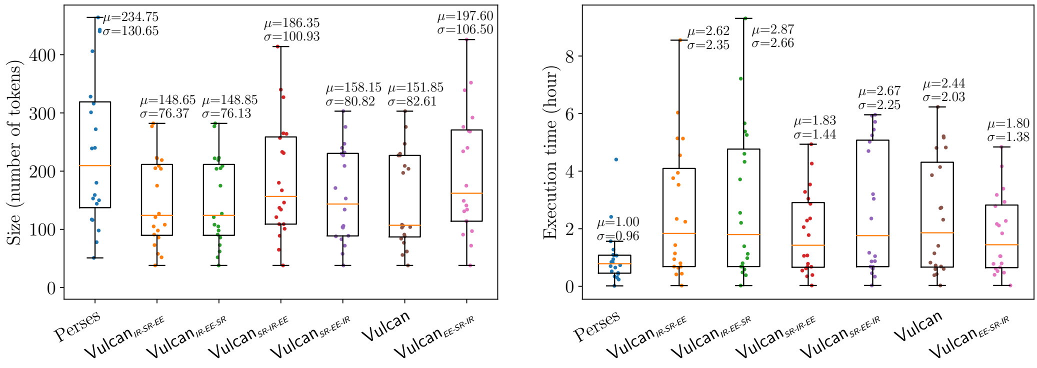
6 DISCUSSION

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

Table 6. Evaluation results of Vulcan and Vulcan^{+} on C program benchmarks. C(%) is the size change percentages of Vulcan^{+}.

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

6.1 Impact of the Order of the Auxiliary Reducers



(a) Sizes of the reduced programs.

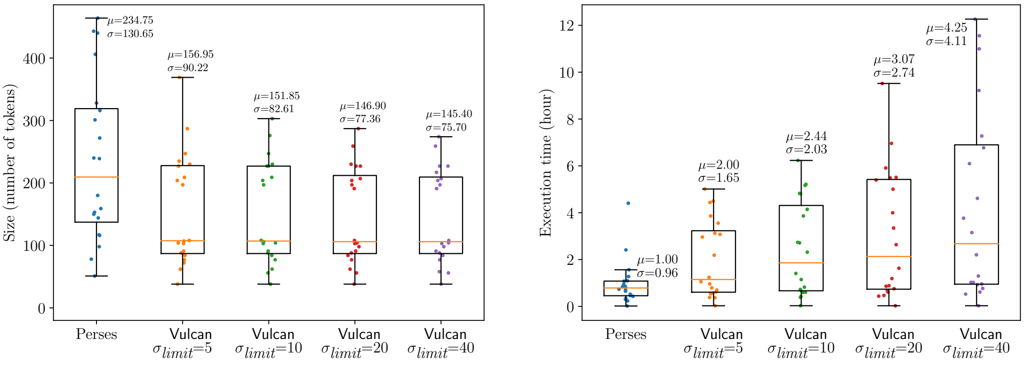
(b) Execution time.

Fig. 6. Evaluation results of Vulcan using different orders of auxiliary reducers on the C benchmarks.

As aforementioned in §4.4, we deduce a relatively reasonable order by analyzing the design goal and computational complexity of each auxiliary reducer and set this order as the default order of Vulcan. In §5, the extensive evaluation results demonstrated the effectiveness of this order. However, it is unclear how Vulcan performs with other orders of the auxiliary reducers. To better understand how the orders of the auxiliary reducers affect the performance, we proposed five variants: Vulcan$_{IR-SR-EE}$, Vulcan$_{IR-EE-SR}$, Vulcan$_{SR-IR-EE}$, Vulcan$_{SR-EE-IR}$, and Vulcan$_{EE-SR-IR}$. Each of them has a different order of the auxiliary reducers from Vulcan, and the subscript indicates the order. We ran Vulcan and all these variants to reduce the 20 C programs.

As shown in Figure 6, all the variants of Vulcan have different performance, but all of them significantly outperform Perses in terms of the output's size. In general, the variant which overall produces smaller results also requires a longer execution time on average. For example, Vulcan$_{IR-SR-EE}$ and Vulcan$_{IR-EE-SR}$ produce smaller results in general than other variants, but they also take relatively longer execution time than other variants. Nevertheless, there are also variants outperformed by other variants in terms of both metrics. For example, compared with Vulcan$_{SR-EE-IR}$, both Vulcan and Vulcan$_{IR-SR-EE}$ produce smaller results while taking a shorter execution time on average. As for the default order that we proposed, its performance in terms of output's average size is comparable to Vulcan$_{IR-SR-EE}$, the best variant on this metric. Meanwhile, the average execution time taken by the default order is relatively short, which is only longer than those of Vulcan$_{SR-IR-EE}$ and Vulcan$_{EE-SR-IR}$, the two most ineffective variants.

6.2 Impact of the Parameters



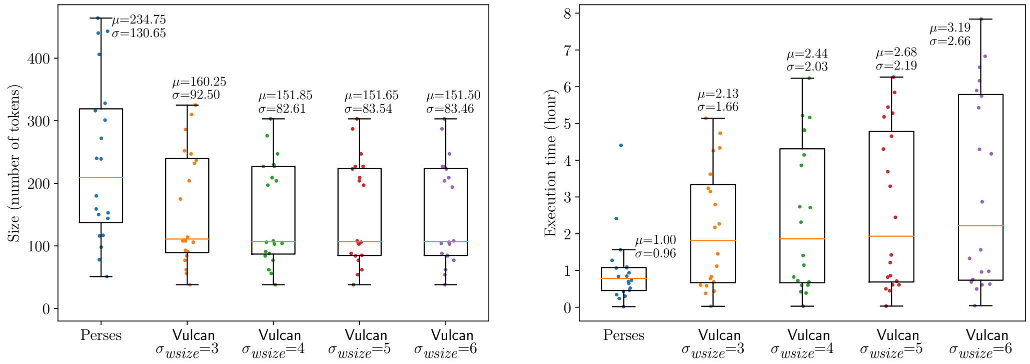
(a) Sizes of the reduced programs.

(b) Execution time.

Fig. 7. Evaluation results of Vulcan with different σ_{limit} on the C benchmarks.

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

To provide more insights for adjusting the parameters, we conducted a parameter study to investigate the impact of the two parameters. In the experiment, we keep one parameter with its default value and set the other parameter with 3 different values. Figure 7 shows the results of running Vulcan on the C benchmarks with different σ_{limit}. As shown in Figure 7a, Vulcan tends to produce smaller results with a higher value of σ_{limit}, though the improvement is moderate. As



(a) Sizes of the reduced programs.

(b) Execution time.

Fig. 8. Evaluation results of Vulcan with different σ_{wsize} on the C benchmarks.

for the execution time, Figure 7b shows the experiment result. In general, as σ_{limit} increases, the required execution time noticeably increases.

The impact of σ_{wsize} is investigated in the same way. As shown in Figure 8, in general, when σ_{wsize} increases, the size of the output decreases, and the execution time increases. However, the overall decrease in size of the result is rather limited, especially when σ_{wsize} is greater than 3.

7 RELATED WORK

AGRs. Delta Debugging (DD) [Zeller and Hildebrandt 2002] is the first work that opens up the research direction on test case reduction. It proposed two algorithms, `dd` and `ddmin`. `dd` can minimize the difference between a failure-inducing test case and a given test case that can be correctly handled. `ddmin` is a special case of `dd`, and it directly minimizes the failure-inducing test case by minimizing the difference between the test case and an empty test case. However, `ddmin` regards code as plain strings of tokens, and thus it does not perform well when the failure-inducing test case is large and structural. To overcome this obstacle, Misherghi and Su proposed Hierarchical Delta Debugging (HDD) [Misherghi and Su 2006], which leverages the tree structure of the test cases to facilitate the reduction process. Specifically, HDD performs `ddmin`, the algorithm proposed in DD, on the parse tree of the test case, from the topmost level to the bottommost level. In this way, HDD can efficiently handle large and highly-structured test cases. Hodován *et al.* also proposed Coarse Hierarchical Delta Debugging (CHDD) to further accelerate the reduction process [Hodován et al. 2017]. Berkeley Delta utilizes *topformflat* to identify nested structures and then perform `ddmin` on them [McPeak et al. 2003]. Despite their effectiveness, none of these aforementioned techniques preserves the syntactical validity of the test case during the reduction process. As a result, when these tools perform program reduction, many syntactical invalid programs are generated and tested against the property. These invalid programs can hardly pass the property tests, and executing property tests on these program wastes considerable time. To solve this problem, Sun *et al.* proposed Perses, a syntax-guided program reduction tool to facilitate the reduction process by promising only to generate and test syntactically valid program [Sun et al. 2018]. Chisel is another program reduction tool designed for program debloating [Heo et al. 2018]. It accelerates the reduction by using a reinforcement learning-based approach. Moreover, ProbDD introduces Bayesian optimization to model the probability of each element appearing in the minimal result, and prioritizes the reduction of those with high probability [Wang et al. 2021].

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

SPRs. In addition to the language-agnostic program reduction tools like HDD and Perses, some reducers only focus on reducing specific languages. For example, C-Reduce is specifically designed to reduce C/C++ programs [Regehr et al. 2012]. It first uses Clang front end to parse the program to be reduced and then invokes a collection of plugins to reduce the program. Each plugin performs a series of well-crafted transformations designed only for C/C++ language. J-Reduce is a reduction tool for Java bytecode proposed by Kalhauge and Palsberg [Kalhauge and Palsberg 2019]. They modeled the task of reducing inputs with many internal dependencies as a problem of dependency graph reduction and proposed a general strategy to solve this problem. Their following work further improved J-Reduce using finer-grained modeling of dependencies [Kalhauge and Palsberg 2021]. JS Delta employs WALA static analysis [IBM 2017] to reduce JavaScript programs [JS Delta 2017]. ddSMT is a program reduction tool for the SMT-LIBv2 format specifically [Niemetz and Biere 2013]. It achieves better performance on inputs in SMT-LIBv2 format by supporting handling the language-specific features, such as macros, named annotations, and scopes defined by `push` and `pop` commands. ddSMT2.0 is a successor of ddSMT [Kremer et al. 2021]. It improves ddSMT by using a hybrid minimization strategy including both the `ddmin`-based strategy used in ddSMT and an orthogonal hierarchical strategy. It also extends ddSMT by supporting the entire family of SMT-LIBv2 language dialects. All these SPRs usually can work effectively for only one specific programming language. In contrast, our framework, Vulcan, is a language-agnostic program reduction tool and can effectively reduce programs in different languages.

8 CONCLUSION

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

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

ACKNOWLEDGMENTS

We thank all the anonymous reviewers in OOPSLA'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

Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/nautilus-fishing-for-deep-bugs-with-grammars/

LLVM Bugzilla. 2016. *Bug LLVM-26760.* Retrieved 2022-10-28 from https://bugs.llvm.org/show_bug.cgi?id=26760

Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated testing of graphics shader compilers. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 93:1–93:29. https://doi.org/10.1145/3133917

GCC-Wiki. 2020. *A guide to Testcase reduction.* Retrieved 2022-09-20 from https://gcc.gnu.org/wiki/A_guide_to_testcase_reduction

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, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 380–394. https://doi.org/10.1145/3243734.3243838

Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2017. Coarse Hierarchical Delta Debugging. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017.* IEEE Computer Society, 194–203. https://doi.org/10.1109/ICSME.2017.26

IBM. 2017. *The T.J. Watson Libraries for Analysis.* Retrieved 2022-10-28 from http://wala.sourceforge.net/

JerryScript. 2022. *Bug Report.* Retrieved 2022-09-20 from https://github.com/jerryscript-project/jerryscript/blob/master/.github/ISSUE_TEMPLATE/bug_report.md

JS Delta. 2017. *JS Delta.* Retrieved 2022-10-28 from https://github.com/wala/jsdelta

Christian Gram Kalhauge and Jens Palsberg. 2019. Binary reduction of dependency graphs. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 556–566. https://doi.org/10.1145/3338906.3338956

Christian Gram Kalhauge and Jens Palsberg. 2021. Logical bytecode reduction. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1003–1016. https://doi.org/10.1145/3453483.3454091

Gereon Kremer, Aina Niemetz, and Mathias Preiner. 2021. ddSMT 2.0: Better Delta Debugging for the SMT-LIBv2 Language and Friends. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12760)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 231–242. https://doi.org/10.1007/978-3-030-81688-9_11

Patrick Kreutzer, Stefan Kraus, and Michael Philippsen. 2020. Language-Agnostic Generation of Compilable Test Programs. In *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020.* IEEE, 39–50. https://doi.org/10.1109/ICST46399.2020.00015

Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. (2014), 216–226. https://doi.org/10.1145/2594291.2594334

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.

LLVM. 2022. *How to Submit an LLVM bug report.* Retrieved 2022-09-20 from https://llvm.org/docs/HowToSubmitABug.html

LLVM/Clang. 2022. Clang documentation – LibTooling. Retrieved 2022-10-28 from https://clang.llvm.org/docs/LibTooling.html

Scott McPeak, Daniel S. Wilkerson, and Simon Goldsmith. 2003. *Berkeley Delta.* Retrieved 2022-10-28 from http://delta.tigris.org/

Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical Delta Debugging. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa (Eds.). ACM, 142–151. https://doi.org/10.1145/1134285.1134307

MozillaSecurity. 2022. *Using Lithium.* Retrieved 2022-10-28 from https://github.com/MozillaSecurity/lithium

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.

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* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 335–346. https://doi.org/10.1145/2254064.2254104

John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2022. *The C-Reducer Github repo.* Retrieved 2022-09-20 from https://github.com/csmith-project/creduce

Rust. 2022. *Rust Issues.* Retrieved 2022-09-20 from https://github.com/rust-lang/rust/issues

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, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 849–863. https://doi.org/10.1145/2983990.2984038

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* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 361–371. https://doi.org/10.1145/3180155.3180236

Guancheng Wang, Ruobing Shen, Junjie Chen, Yingfei Xiong, and Lu Zhang. 2021. Probabilistic Delta debugging. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 881–892. https://doi.org/10.1145/3468264.3468625

Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 724–735. https://doi.org/10.1109/ICSE.2019.00081

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* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

A. Zeller and R. 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

Received 2022-10-28; accepted 2023-02-25